elasticsearch.
权威指南

Table of Contents

Elasticsearch 权威指南

项目信息

在线阅读

国外自动指向 GITBOOK 项目 | 国内用户自动指向 阿里云

GITHUB 仓库

译者前言

译者现在的工作项目中需要用到 Elasticsearch，但是在网络中找了很多的相关的内容都很不完善，中文的文档更是寥寥无几，所以我决定边研究边翻译一下官方推出的权威手册。在这里要先感谢原作者们！如果各位在这里发现了错误之处，请大家在 Issue 中提出或者 PR 这个项目。

> 如果你喜欢这个翻译项目可以点击**Star**一下，您的支持是我们最大的动力。

原作名字：elasticsearch - the definitive guide

原作作者：clinton gormley，zachary tong

译者：Gavin Foo fuxiaopang@gmail.com

前言

这本书还在不断地添加内容中，我们会陆陆续续地在这里添加新的章节。这本书中的内容针对的是 Elasticsearch 的最新版本。

欢迎反馈 – 如果这里出现了错误，或者你有什么建议可以到我们 GitHub 项目中 新建一个 issue。

这个世界已经被数据淹没。我们创造的系统所产生的数据可以瞬间轻而易举地将我们压垮，现有的科技一直致力于如何存储数据，并能将拥有大量信息的数据仓库结构化。而当你准备开始从大量的数据中得出结论做决策的时候，美好的一天就要被毁灭了……

Elasticsearch 是一个分布式可扩展的实时搜索和分析引擎。它能帮助你搜索、分析和浏览数据，而往往大家并没有在某个项目一开始就预料到需要这些功能。Elasticsearch 之所以出现就是为了重新赋予硬盘中看似无用的原始数据新的活力。

无论你是需要全文搜索、结构化数据的实时统计，还是两者的结合，这本指南都会帮助你了解其中最基本的概念，从最基本的操作开始学习 Elasticsearch。之后，我们还会逐渐开始探索更加复杂的搜索技术，你可以根据自身的学习的步伐。

Elasticsearch 并不是单纯的全文搜索这么简单。我们将向你介绍讲解结构化搜索、统计、查询过滤、地理定位、自动完成以及你是不是要查找的提示。我们还将探讨如何给数据建模能提升 Elasticsearch 的性能，以及在生产环境中如何配置、监视你的集群。

入门

Elasticsearch 是一个实时的分布式搜索和分析引擎。它可以帮助你用前所未有的速度去处理大规模数据。

它可以用于全文搜索，结构化搜索以及分析，当然你也可以将这三者进行组合

- 维基百科使用 Elasticsearch 来进行全文搜索并高亮显示关键词，以及提供search-as-you-type、did-you-mean等搜索建议功能。
- 英国卫报使用 Elasticsearch 来处理访客日志，以便能将公众对不同文章的反应实时地反馈给各位编辑。
- StackOverflow 将全文搜索与地理位置和相关信息进行结合，以提供more-like-this相关问题的展现。
- GitHub 使用 Elasticsearch 来检索超过1300亿行代码。
- 每天，Goldman Sachs 使用它来处理5TB数据的索引，还有很多投行使用它来分析股票市场的变动。

但是Elasticsearch并不只是面向大型企业的，它还帮助了很多类似 DataDog 以及 Klout 的创业公司进行了功能的扩展。Elasticsearch 可以运行在你的笔记本上，也可以部署到成千上万的服务器上，处理PB级别的数据。

Elasticsearch 每一个独立的部分都不是新创的。比如全文搜索早就已经被实现，统计系统和分布式数据库也早已存在。但是革命之处在于能将这些独立的功能结合成一个连贯、实时处理的整体。对于新用户，它的门槛也很低，当然他也会因为你的强大而变得更强大。

你之所以拿起这本书，就是因为你眼前有很多的数据，但是你并不知道如何使用他们，接下来我们将开始探讨有关处理数据的事情。

很不幸的是，目前的大部分数据库在提取数据方面都是非常的薄弱的。虽然它们可以通过精准的时间戳或者确切的数值来进行内容的筛选，但是它们可以在全文搜索时做到同义词或者相关性搜索吗？他们可以汇总相同内容数据吗？最重要的是，每对如此巨大的数据量，它们能做到实时处理吗？

这便是 Elasticsearch 如此突出的理由：Elasticsearch 可以帮助你浏览并利用已经快要烂在数据库里的那些极难查询的数据。

> **Elasticsearch** 将会成为你一生的小伙伴。

You Know, for Search...

Elasticsearch 是一个建立在全文搜索引擎框架 Apache Lucene(TM) 基础上的开源搜索引擎，无论是开源软件还是私有软件，Lucene 都毫无疑问是当今最先进、性能最高和功能最全的搜索引擎框架。

但是 Lucene 只是一个框架，要充分使用它的功能，你需要使用 JAVA 作为开发语言，并且在你的程序中集成 Lucene。更糟的是，你需要深入了解相关知识才能明白它是如何运行的，Lucene 确实非常复杂。

Elasticsearch 也是使用 Java 编写的，并且采用了 Lucene 来实现索引与搜索的功能。而且，在你使用它做全文搜索时，只需要使用简单流畅的 RESTful API 即可，并不需要了解 Lucene 背后复杂的的运行原理。

当然 Elasticsearch 还有很多地方超越了 Lucene，它不仅可以实现全文搜索功能，还可以完成以下工作:

- 分布式实时文档存储，并将每一个字段都编入索引，使其可以被搜索。
- 分布式实时分析与搜索引擎。
- 可以扩展到上百台服务器，处理PB级别的结构化或非结构化数据。

这么多的功能被集成到一台服务器上，你可以轻松地通过客户端或者任何你喜欢的程序语言与 Elasticsearch 的 RESTful API 进行通信。

Elasticsearch 的上手是非常简单的。它附带了很多非常合理的默认值，这让初学者很好地避免一上手就要面对复杂的理论。安装完成就可以马上投入使用了。不需要了解很多，你就能变得非常有生产力。

随着学习的深入，你还可以使用 Elasticsearch 更多高级的功能。整个引擎可以很灵活地进行配置。你可以根据自身需求来定制属于你自己的 Elasticsearch。

你可以随意下载、使用、修改 Elasticsearch。它采用 Apache 2 license 进行授权，这是当前最灵活的开源授权方式。源代码托管在 Github 之上：github.com/elastic/elasticsearch Elasticsearch 在 Apache 2 license下许可使用，可以免费下载、使用和修改。如果你愿意加入我们非常优秀的贡献者社区，请参考：Elasticsearch 贡献。

如果你对 Elasticsearch 的功能、插件、SDK 等任何方面有疑问，都可以在这里提出 discuss.elastic.co，elastic 中文社区。

回忆时光

多年以前，有个叫 Shay Banon 的失业开发者跟随他的新婚妻子来到了伦敦，因为他妻子将在那里学习厨艺。Shay 在寻找工作的同时，开始研究还是早期版本的 Lucene，并打算为她妻子制作一个烹饪菜谱搜索引擎。

直接基于 Lucene 工作会比较困难，所以 Shay 开始实现一个 Lucene 之上的抽象层，这样 Java 程序员可以很方便的为他们的应用程序添加搜索功能。于是他发布了自己的第一个开源项目 “Compass”。

后来 Shay 找到了一份工作，这份工作主要围绕在高性能与分布式内存数据存储的环境中。高性能、实时、分布式搜索引擎是必不可少的需求。因此他决定重写 Compass 库，使其成为一个独立的服务器，这便是 Elasticsearch。

2010年的2月份，第一个公开版本发布了。从此之后，Elasticsearch 成为 Github 上最受欢迎的项目之一，超过300个代码贡献者。一家关于 Elasticsearch 的公司就此成立，它们不仅提供商业支持还在进行新功能的开发，但是 Elasticsearch 一定会永远向大众开放，永远开源给所有需要的人们。

噢，对了，Shay 的妻子还在等待着她的菜谱搜索引擎。

安装 JAVA

```
yum install java-1.7.0-openjdk -y
```

安装并运行 Elasticsearch

了解 Elasticsearch 最简单的方法就是去使用它，让我们一起开始探索之旅吧！

安装 Elasticsearch 只有一个要求，那就是需要安装最新版本的 Java。而且最好从 Java 官网 www.java.com 下载最新版本的 Java 来安装。

你可以从这里获取到最新版本的Elasticsearch：elastic.co/downloads/elasticsearch。

要安装 Elasticsearch，你需要下载并解压对应运行平台的压缩文件。更多相关信息请参考 Elasticsearch Reference。

TIP

```
当你在生产环境中安装 Elasticsearch 时，你可以选择使用 Debian 或者 RPM 的安装包，地址如下：[downl
oads page](http://www.elastic.co/downloads/elasticsearch)。你也可以使用官方提供的 [Puppet
module](https://github.com/elasticsearch/puppet-elasticsearch) 或者 [Chef cookbook](htt
ps://github.com/elasticsearch/cookbook-elasticsearch)。
```

解压完成后，Elasticsearch 就已经准备就绪，等待运行了。执行以下命令便可在前台启动它：

```
cd elasticsearch-<version>
./bin/elasticsearch <1> <2>
```

1. 如果你想在后台以守护进程模式运行它，请添加 `-d` 参数。
2. 如果你是在 Windows 中运行 Elasticsearch，只需要运行 `bin\elasticsearch.bat` 即可。

你可以在另一个终端窗口中运行以下命令来验证它是否成功运行：

```
curl 'http://localhost:9200/?pretty'
```

TIP: 如果是在 Windows 中运行 Elasticsearch，你可以从 http://curl.haxx.se/download.html 下载 cURL。安装后，遍可以通过 cURL 简单地向 Elasticsearch 提交请求。并且你也可以直接复制粘贴本书中众多的例子，通过 cURL 来运行与试验。

你会看到如下的返回信息：

```
{
  "name" : "Tom Foster",
  "cluster_name" : "elasticsearch",
  "version" : {
    "number" : "2.1.0",
    "build_hash" : "72cd1f1a3eee09505e036106146dc1949dc5dc87",
    "build_timestamp" : "2015-11-18T22:40:03Z",
    "build_snapshot" : false,
    "lucene_version" : "5.3.1"
  },
  "tagline" : "You Know, for Search"
}
```

SENSE: 010_Intro/10_Info.json

这说明你已经开启并运行了一个 ELasticsearch 节点，接下来就可以开始各种实验了。节点（*node*） 是 Elasticsearch 中的一个运行实例。集群（*cluster*） 是一个包含了多个拥有相同 `cluster.name` 节点的分组，这些节点协同工作以共享数据，并且提供了故障转移以及扩展的可能性。（一个节点也可以构成一个集群。）你可以在配置文件 `elasticsearch.yml` 中修改 `cluster.name`，每当节点启动时，这些配置文件就会被加载。更多相关信息可以参考本书最后关于生产部署部分的 《important-configuration-changes, 重要的配置修改》章节。

TIP: 看到例子下方的 View in Sense 链接了吗？ 《sense, 安装 Sense 控制台》 便可以在自己的 Elasticsearch 集群中运行本书中的例子，并直接查看结果了。

当 Elasticsearch 已经在前台运行时，你可以按下 Ctrl-C 来终止这个进程。

安装 Sense

Sense 是一个 Kibana 程序，它的交互式控制台可以帮助你直接通过浏览器向 Elasticsearch 提交请求。 在本书的在线版中，众多的代码示例都包含了 View in Sense 链接。当你点击之后，它将自动在 Sense 控制台中运行这段代码。你并不是一定要安装 Sense，但那将失去很多与本书的互动以及直接在你本地的集群中的实验代码的乐趣。

安装并运行 Sense:

在 Kibana 的目录中运行以下命令以下载并安装 Sense 程序:

```
./bin/kibana plugin --install elastic/sense <1>
```

1. Windows: `bin\kibana.bat plugin --install elastic/sense` .

NOTE: 如果需要离线安装 Sense，你可以从这里直接下载：
https://download.elastic.co/elastic/sense/sense-latest.tar.gz。

运行 Kibana.

```
./bin/kibana <1>
```

1. Windows: `bin\kibana.bat` .

在浏览器中访问 `http://localhost:5601/app/sense` 就可以使用 Sense 了。

与 Elasticsearch 通信

如何与 Elasticsearch 通信要取决于你是否使用 JAVA。

Java API

如果你使用的是 JAVA，Elasticsearch 内置了两个客户端，你可以在你的代码中使用：

节点客户端: 节点客户端以一个 无数据节点 的身份加入了一个集群。换句话说，它自身是没有任何数据的，但是他知道什么数据在集群中的哪一个节点上，然后就可以请求转发到正确的节点上并进行连接。

传输客户端: 更加轻量的传输客户端可以被用来向远程集群发送请求。他并不加入集群本身，而是把请求转发到集群中的节点。

这两个客户端都使用 Elasticsearch 的 传输 协议，通过**9300**端口与 java 客户端进行通信。集群中的各个节点也是通过9300端口进行通信。如果这个端口被禁止了，那么你的节点们将不能组成一个集群。

> ## TIP

Java 的客户端的版本号必须要与 Elasticsearch 节点所用的版本号一样，不然他们之间可能无法识别。

更多关于 Java API 的说明可以在这里找到 Guide.

通过 HTTP 向 RESTful API 传送 json

其他的语言可以通过*9200*端口与 Elasticsearch 的 RESTful API 进行通信。事实上，如你所见，你甚至可以使用行命令 `curl` 来与 Elasticsearch 通信。

Elasticsearch 官方提供了很多种编程语言的客户端，也有和许多社区化软件的集成插件，这些都可以在 Guide 里面找到。

向 Elasticsearch 发出的请求和其他所有的 HTTP 请求的组成部分是一致的。例如，计算集群中文件的数量，我们就可以使用：

```
      <1>     <2>                     <3>    <4>
curl -XGET 'http://localhost:9200/_count?pretty' -d '
{  <5>
    "query": {
        "match_all": {}
    }
}
'
```

1. 相应的 HTTP 请求方法 或者 变量： `GET` , `POST` , `PUT` , `HEAD` 或者 `DELETE` 。
2. 集群中任意一个节点的访问协议、主机名以及端口。
3. 请求的路径。
4. 任意一个查询后再加上 `?pretty` 就可以生成 更加美观 的JSON反馈，以增强可读性。
5. 一个 JSON 编码的请求主体（如果需要的话）。

Elasticsearch 将会返回一个 HTTP 状态码类似于 '200 OK'，以及一个 JSON 格式的主体（除了单纯的 'HEAD' 请求），上面的请求会得到下方的 JSON 主体：

```
{
    "count" : 0,
    "_shards" : {
        "total" : 5,
        "successful" : 5,
        "failed" : 0
    }
}
```

在反馈中，我们并没有看见 HTTP 的头部信息，因为我们没有告知 `curl` 显示这些内容。如果你想看到头部信息，可以在使用 `curl` 命令的时候再加上 `-i` 这个参数：

```
curl -i -XGET 'localhost:9200/'
```

从现在开始，本书里所有涉及 `curl` 命令的部分我们都会进行简写，因为主机、端口等信息都是相同的，缩减前的样子:

```
curl -XGET 'localhost:9200/_count?pretty' -d '
{
    "query": {
        "match_all": {}
    }
}'
```

我们将会简写成这样:

```
GET /_count
{
    "query": {
        "match_all": {}
    }
}
```

面向文档

程序中的对象很少是单纯的键值与数值的列表。更多的时候它拥有一个复杂的结构，比如包含了日期、地理位置、对象、数组等。

迟早你会把这些对象存储在数据库中。你会试图将这些丰富而又庞大的数据都放到一个由行与列组成的关系数据库中，然后你不得不根据每个字段的格式来调整数据，然后每次重建它你都要检索一遍数据。

Elasticsearch 是 面向文档型数据库，这意味着它存储的是整个对象或者 文档，它不但会存储它们，还会为他们建立索引，这样你就可以搜索他们了。你可以在 Elasticsearch 中索引、搜索、排序和过滤这些文档。不需要成行成列的数据。这将会是完全不同的一种面对数据的思考方式，这也是为什么 Elasticsearch 可以执行复杂的全文搜索的原因。

JSON

Elasticsearch使用 *JSON* (或称作JavaScript Object Notation) 作为文档序列化的格式。JSON 已经被大多数语言支持，也成为 NoSQL 领域的一个标准格式。它简单、简洁、易于阅读。

把这个 JSON 想象成一个用户对象:

```
{
    "email":      "john@smith.com",
    "first_name": "John",
    "last_name":  "Smith",
    "about": {
        "bio":         "Eco-warrior and defender of the weak",
        "age":         25,
        "interests": [ "dolphins", "whales" ]
    },
    "join_date": "2014/05/01",
}
```

虽然 `user` 这个对象非常复杂，但是它的结构和含义都被保留到 JSON 中了。在 Elasticsearch 中，将对象转换为 JSON 并作为索引要比在表结构中做相同的事情简单多了。

> 将你的数据转换为 **JSON**

几乎所有的语言都有将任意数据转换、机构化成 JSON，或者将对象转换为JSON的模块。查看 `serialization` 以及 `marshalling` 两个 JSON 模块。The official Elasticsearch clients 也可以帮你自动结构化 JSON。

启程

为了能让你感受一下 Elasticsearch 能做什么以及它是有多么的易用，我们会先为你简单展示一下，其中包括了基本的 创建索引，搜索 以及 聚合。

我们会在这里向你介绍一些新的术语以及简单的概念，即使你没有马上接受这些概念也没有关系。我们会在之后的章节中逐渐帮你理解它们。

所以，准备开始享受 Elasticsearch 的学习之旅把！

建立一个员工名单

想象我们正在为一个名叫 megacorp 的公司的 HR 部门制作一个新的员工名单系统，这些名单应该可以满足实时协同工作，所以它应该可以满足以下要求：

- 数据可以包含多个值的标签、数字以及纯文本内容，
- 可以检索任何职员的所有数据。
- 允许结构化搜索。例如，查找30岁以上的员工。
- 允许简单的全文搜索以及相对复杂的短语搜索。
- 在返回的匹配文档中高亮关键字。
- 拥有数据统计与管理的后台。

为员工档案创建索引

这个项目的第一步就是存储员工的数据。这样你就需要一个“员工档案”的表单，这样每个文档都代表着一个员工。在 Elasticsearch 中，存储数据的行为就叫做 索引*(indexing)* ，但是在我们索引数据前，我们需要决定将数据存储在哪里。

在 Elasticsearch 中，文档属于一种 类型*(type)*，各种各样的类型存在于一个 索引 中。你也可以通过类比传统的关系数据库得到一些大致的相似之处：

```
关系数据库      ⇒ 数据库 ⇒ 表    ⇒ 行    ⇒ 列(Columns)
Elasticsearch  ⇒ 索引   ⇒ 类型  ⇒ 文档  ⇒ 字段(Fields)
```

一个 Elasticsearch 集群可以包含多个 索引（数据库），也就是说其中包含了很多 类型（表）。这些类型中包含了很多的 文档（行），然后每个文档中又包含了很多的 字段（列）。

> 索引 索引 索引

你可能发现在 Elasticsearch 中，索引这个词汇已经被赋予了太多意义，所以在这里我们有必要澄清一下：

> 索引 (名词)

如上文所说，一个 索引 就类似于传统关系型数据库中的 数据库。这里就是存储相关文档的的地方。

> 索引 (动词)

为一个文档创建索引 是把一个文档存储到一个索引*(名词)*中的过程，这样它才能被检索。这个过程非常类似于 SQL 中的 `INSERT` 命令，如果已经存在文档，新的文档将会覆盖旧的文档。

> 反向索引

在关系数据库中的某列添加一个 索引，比如多路搜索树(B-Tree)索引，就可以加速数据的取回速度， Elasticsearch 以及 Lucene 使用的是一个叫做 反向索引*(inverted index)* 的结构来实现相同的功能。

> 通常，每个文档中的字段都被创建了索引（拥有一个反向索引），因此他们可以被搜索。如果一个字段缺失了反向索引的话，它将不能被搜索。我们将会在之后的《反向索引》章节中详细介绍它。

所以为了创建员工名单，我们需要进行如下操作：

- 为每一个员工的 文档 创建索引，每个 文档 都包含了一个员工的所有信息。
- 每个文档都会被标记为 `employee` 类型。
- 这种类型将存活在 `megacorp` 这个 索引 中。
- 这个索引将会存储在 Elasticsearch 的集群中

在实际的操作中，这些操作是非常简单的（即使看起来有这么多步骤）。我们可以把如此之多的操作通过一个命令来完成：

```
PUT /megacorp/employee/1
{
    "first_name" : "John",
    "last_name" :  "Smith",
    "age" :        25,
    "about" :      "I love to go rock climbing",
    "interests": [ "sports", "music" ]
}
```

注意在 `/megacorp/employee/1` 路径下，包含了三个部分：

名字	内容
megacorp	索引的名字
employee	类型的名字
1	当前员工的ID

请求部分，也就是 JSON 文档，在这里包含了关于这名员工的所有信息。他的名字是 “John Smith”，他已经25岁了，他很喜欢攀岩。

怎么样？很简单吧！我们在操作前不需要进行任何管理操作，比如创建索引，或者为字段指定数据的类型。我们就这么直接地为一个文档创建了索引。Elasticsearch 会在创建的时候为它们设定默认值，所以所有管理操作已经在后台被默默地完成了。

在进行下一步之前，我们再为这个目录添加更多的员工信息吧：

```
PUT /megacorp/employee/2
{
    "first_name" :  "Jane",
    "last_name" :   "Smith",
    "age" :         32,
    "about" :       "I like to collect rock albums",
    "interests":  [ "music" ]
}

PUT /megacorp/employee/3
{
    "first_name" :  "Douglas",
    "last_name" :   "Fir",
    "age" :         35,
    "about":        "I like to build cabinets",
    "interests":  [ "forestry" ]
}
```

检索文档

现在，我们已经在 Elasticsearch 中存储了一些数据，我们可以开始根据这个项目的需求进行工作了。第一个需求就是要能搜索每一个员工的数据。

对于 Elasticsearch 来说，这是非常简单的。我们只需要执行一次 HTTP GET 请求，然后指出文档的地址，也就是索引、类型以及 ID 即可。通过这三个部分，我们就可以得到原始的 JSON 文档：

```
GET /megacorp/employee/1
```

返回的内容包含了这个文档的元数据信息，而 John Smith 的原始 JSON 文档也在 `_source` 字段中出现了：

```
{
  "_index" :   "megacorp",
  "_type" :    "employee",
  "_id" :      "1",
  "_version" : 1,
  "found" :    true,
  "_source" :  {
      "first_name" :  "John",
      "last_name" :   "Smith",
      "age" :         25,
      "about" :       "I love to go rock climbing",
      "interests":  [ "sports", "music" ]
  }
}
```

我们通过将HTTP后的请求方式由 `PUT` 改变为 `GET` 来获取文档，同理，我们也可以将其更换为 `DELETE` 来删除这个文档，`HEAD` 是用来查询这个文档是否存在的。如果你想替换一个已经存在的文档，你只需要使用 `PUT` 再次发出请求即可。

简易搜索

`GET` 命令真的相当简单，你只需要告诉它你要什么即可。接下来，我们来试一下稍微复杂一点的搜索。

我们首先要完成一个最简单的搜索命令来搜索全部员工：

```
GET /megacorp/employee/_search
```

你可以发现我们正在使用 `megacorp` 索引，`employee` 类型，但是我们我们并没有指定文档的ID，我们现在使用的是 `_search` 端口。你可以再返回的 `hits` 中发现我们录入的三个文档。搜索会默认返回最前的10个数值。

```
{
   "took":      6,
   "timed_out": false,
   "_shards": { ... },
   "hits": {
      "total":      3,
      "max_score":  1,
      "hits": [
         {
            "_index":         "megacorp",
            "_type":          "employee",
            "_id":            "3",
            "_score":         1,
            "_source": {
               "first_name":  "Douglas",
               "last_name":   "Fir",
               "age":         35,
               "about":       "I like to build cabinets",
               "interests": [ "forestry" ]
            }
         },
         {
            "_index":         "megacorp",
            "_type":          "employee",
            "_id":            "1",
            "_score":         1,
            "_source": {
               "first_name":  "John",
               "last_name":   "Smith",
               "age":         25,
               "about":       "I love to go rock climbing",
               "interests": [ "sports", "music" ]
            }
         },
         {
            "_index":         "megacorp",
            "_type":          "employee",
            "_id":            "2",
            "_score":         1,
            "_source": {
               "first_name":  "Jane",
               "last_name":   "Smith",
               "age":         32,
               "about":       "I like to collect rock albums",
               "interests": [ "music" ]
            }
         }
      ]
   }
}
```

注意：反馈值中不仅会告诉你匹配到哪些文档，同时也会把这个文档都会包含到其中：我们需要搜索的用户的所有信息。

接下来，我们将要尝试着实现搜索一下哪些员工的姓氏中包含 `Smith` 。为了实现这个，我们需要使用一种轻量的搜索方法。这种方法经常被称做 查询字符串*(query string)* 搜索，因为我们通过URL来传递查询的关键字：

```
GET /megacorp/employee/_search?q=last_name:Smith
```

我们依旧使用 `_search` 端口，然后可以将参数传入给 `q=` 。这样我们就可以得到姓Smith的结果：

```
{
   ...
   "hits": {
      "total":      2,
      "max_score":  0.30685282,
      "hits": [
         {
            ...
            "_source": {
               "first_name":  "John",
               "last_name":   "Smith",
               "age":         25,
               "about":       "I love to go rock climbing",
               "interests": [ "sports", "music" ]
            }
         },
         {
            ...
            "_source": {
               "first_name":  "Jane",
               "last_name":   "Smith",
               "age":         32,
               "about":       "I like to collect rock albums",
               "interests": [ "music" ]
            }
         }
      ]
   }
}
```

使用Query DSL搜索

查询字符串是通过命令语句完成 点对点*(ad hoc)* 的搜索，但是这也有它的局限性（可参阅《搜索局限性》章节）。Elasticsearch 提供了更加丰富灵活的查询语言，它被称作 *Query DSL*，通过它你可以完成更加复杂、强大的搜索任务。

DSL (*Domain Specific Language* 领域特定语言) 需要使用 JSON 作为主体，我们还可以这样查询姓 Smith 的员工：

```
GET /megacorp/employee/_search
{
    "query" : {
        "match" : {
            "last_name" : "Smith"
        }
    }
}
```

这个请求会返回同样的结果。你会发现我们在这里没有使用 查询字符串，而是使用了一个由 JSON 构成的请求体，其中使用了 `match` 查询法，随后我们还将会学习到其他的查询类型。

更加复杂的搜索

接下来，我们再提高一点儿搜索的难度。我们依旧要寻找出姓 Smith 的员工，但是我们还将添加一个年龄大于30岁的限定条件。我们的查询语句将会有一些细微的调整来以识别结构化搜索的限定条件 *filter*（过滤器）:

```
GET /megacorp/employee/_search
{
    "query" : {
        "filtered" : {
            "filter" : {
                "range" : {
                    "age" : { "gt" : 30 } <1>
                }
            },
            "query" : {
                "match" : {
                    "last_name" : "Smith" <2>
                }
            }
        }
    }
}
```

1. 这一部分的语句是 `range` *filter* ，它可以查询所有超过30岁的数据 -- `gt` 代表 **greater than** （大于）。

2. 这一部分我们前一个操作的 `match` *query* 是一样的

先不要被这么多的语句吓到，我们将会在之后带你逐渐了解他们的用法。你现在只需要知道我们添加了一个 *filter*，可以在 `match` 的搜索基础上再来实现区间搜索。现在，我们的只会显示32岁的名为**Jane Smith**的员工了：

```
{
   ...
   "hits": {
      "total":      1,
      "max_score":  0.30685282,
      "hits": [
         {
            ...
            "_source": {
               "first_name":  "Jane",
               "last_name":   "Smith",
               "age":         32,
               "about":       "I like to collect rock albums",
               "interests": [ "music" ]
            }
         }
      ]
   }
}
```

全文搜索

上面的搜索都很简单：名字搜索、通过年龄过滤。接下来我们来学习一下更加复杂的搜索，全文搜索——一项在传统数据库很难实现的功能。 我们将会搜索所有喜欢 **rock climbing** 的员工:

```
GET /megacorp/employee/_search
{
    "query" : {
        "match" : {
            "about" : "rock climbing"
        }
    }
}
```

你会发现我们同样使用了 `match` 查询来搜索 `about` 字段中的 **rock climbing**。我们会得到两个匹配的文档：

```
{
   ...
   "hits": {
      "total":      2,
      "max_score":  0.16273327,
      "hits": [
         {
            ...
            "_score":         0.16273327, <1>
            "_source": {
               "first_name":  "John",
               "last_name":   "Smith",
               "age":         25,
               "about":       "I love to go rock climbing",
               "interests": [ "sports", "music" ]
            }
         },
         {
            ...
            "_score":         0.016878016, <1>
            "_source": {
               "first_name":  "Jane",
               "last_name":   "Smith",
               "age":         32,
               "about":       "I like to collect rock albums",
               "interests": [ "music" ]
            }
         }
      ]
   }
}
```

1. 相关评分

通常情况下，Elasticsearch 会通过相关性来排列顺序，第一个结果中，John Smith 的 `about` 字段中明确地写到 **rock climbing**。而在 Jane Smith 的 `about` 字段中，提及到了 **rock**，但是并没有提及到 **climbing**，所以后者的 `_score` 就要比前者的低。

这个例子很好地解释了 Elasticsearch 是如何执行全文搜索的。对于 Elasticsearch 来说，相关性的概念是很重要的，而这也是它与传统数据库在返回匹配数据时最大的不同之处。

段落搜索

能够找出每个字段中的独立单词固然很好，但是有的时候你可能还需要去匹配精确的短语或者 段落。例如，我们只需要查询到 `about` 字段只包含 **rock climbing** 的短语的员工。

为了实现这个效果，我们将对 `match` 查询变为 `match_phrase` 查询：

```
GET /megacorp/employee/_search
{
    "query" : {
        "match_phrase" : {
            "about" : "rock climbing"
        }
    }
}
```

这样，系统会没有异议地返回 John Smith 的文档：

```
{
   ...
   "hits": {
      "total":      1,
      "max_score":  0.23013961,
      "hits": [
         {
            ...
            "_score":         0.23013961,
            "_source": {
               "first_name":  "John",
               "last_name":   "Smith",
               "age":         25,
               "about":       "I love to go rock climbing",
               "interests": [ "sports", "music" ]
            }
         }
      ]
   }
}
```

高亮我们的搜索

很多程序希望能在搜索结果中 高亮 匹配到的关键字来告诉用户这个文档是 如何 匹配他们的搜索的。在 Elasticsearch 中找到高亮片段是非常容易的。

让我们回到之前的查询，但是添加一个 `highlight` 参数：

```
GET /megacorp/employee/_search
{
    "query" : {
        "match_phrase" : {
            "about" : "rock climbing"
        }
    },
    "highlight": {
        "fields" : {
            "about" : {}
        }
    }
}
```

当我们运行这个查询后，相同的命中结果会被返回，但是我们会得到一个新的名叫 `highlight` 的部分。在这里包含了 `about` 字段中的匹配单词，并且会被 `<em></em>` HTML 字符包裹住：

```
{
   ...
   "hits": {
      "total":      1,
      "max_score":  0.23013961,
      "hits": [
         {
            ...
            "_score":         0.23013961,
            "_source": {
               "first_name":  "John",
               "last_name":   "Smith",
               "age":         25,
               "about":       "I love to go rock climbing",
               "interests": [ "sports", "music" ]
            },
            "highlight": {
               "about": [
                  "I love to go <em>rock</em> <em>climbing</em>" <1>
               ]
            }
         }
      ]
   }
}
```

1. 在原有文本中高亮关键字。

统计

最后，我们还有一个需求需要完成：可以让老板在职工目录中进行统计。Elasticsearch 把这项功能称作 汇总 *(aggregations)*，通过这个功能，我们可以针对你的数据进行复杂的统计。这个功能有些类似于 SQL 中的 `GROUP BY` ，但是要比它更加强大。

例如，让我们找一下员工中最受欢迎的兴趣是什么：

```
GET /megacorp/employee/_search
{
  "aggs": {
    "all_interests": {
      "terms": { "field": "interests" }
    }
  }
}
```

请忽略语法，让我们先来看一下结果：

```
{
   ...
   "hits": { ... },
   "aggregations": {
      "all_interests": {
         "buckets": [
            {
               "key":       "music",
               "doc_count": 2
            },
            {
               "key":       "forestry",
               "doc_count": 1
            },
            {
               "key":       "sports",
               "doc_count": 1
            }
         ]
      }
   }
}
```

我们可以发现有两个员工喜欢音乐，还有一个喜欢森林，还有一个喜欢运动。这些数据并没有被预先计算好，它们是在文档被查询的同时实时计算得出的。如果你想要查询姓 Smith 的员工的兴趣汇总情况，你就可以执行如下查询：

```
GET /megacorp/employee/_search
{
  "query": {
    "match": {
      "last_name": "smith"
    }
  },
  "aggs": {
    "all_interests": {
      "terms": {
        "field": "interests"
      }
    }
  }
}
```

这样， `all_interests` 的统计结果就只会包含满足查询的文档了：

```
  ...
  "all_interests": {
     "buckets": [
        {
           "key": "music",
           "doc_count": 2
        },
        {
           "key": "sports",
           "doc_count": 1
        }
     ]
  }
```

汇总还允许多个层面的统计。比如我们还可以统计每一个兴趣下的平均年龄：

```
GET /megacorp/employee/_search
{
    "aggs" : {
        "all_interests" : {
            "terms" : { "field" : "interests" },
            "aggs" : {
                "avg_age" : {
                    "avg" : { "field" : "age" }
                }
            }
        }
    }
}
```

虽然这次返回的汇总结果变得更加复杂了，但是它依旧很容易理解：

```
...
"all_interests": {
   "buckets": [
      {
         "key": "music",
         "doc_count": 2,
         "avg_age": {
            "value": 28.5
         }
      },
      {
         "key": "forestry",
         "doc_count": 1,
         "avg_age": {
            "value": 35
         }
      },
      {
         "key": "sports",
         "doc_count": 1,
         "avg_age": {
            "value": 25
         }
      }
   ]
}
```

在这个丰富的结果中，我们不但可以看到兴趣的统计数据，还能针对不同的兴趣来分析喜欢这个兴趣的 `平均年龄` 。

即使你现在还不能很好地理解语法，但是相信你还是能发现，用这个功能来实现如此复杂的统计工作是这样的简单。你的极限取决于你存入了什么样的数据哟！

小结

希望上面的几个小教程可以很好地向你解释 Elasticsearch 可以实现什么功能。为了保持教程简短，这里只提及了一些基础，除此之外还有很多功能，比如建议、地理定位、过滤、模糊以及部分匹配等。但是相信你也发现了，在这里你只需要很简单的操作就可以完成复杂的操作。无需配置，添加数据就可以开始搜索！

可能前面有一些语法会让你觉得很难理解，你可能对如何调整优化它们还有很多疑问。那么，本书之后的章节将会帮助你逐步解开疑问，让你对 Elasticsearch 是如何工作的有一个全面的了解。

分布式特性

在最开始的章节中，我们曾经提到 Elasticsearch 可以被扩展到上百台（甚至上千台）服务器上，来处理PB级别的数据。我们的教程只提及了如何使用它，但是并没有提及到服务器方面的内容。Elasticsearch 是自动分布的，它在设计时就考虑到可以隐藏分布操作的复杂性。

Elasticsearch 的分布式部分很简单。你甚至不需要关于分布式系统的任何内容，比如分片、集群、发现等成堆的分布式概念。你可能在你的笔记本中运行着刚才的教程，如果你想在一个拥有100个节点的集群中运行教程，你会发现操作是完全一样的。

Elasticsearch 很努力地在避免复杂的分布式系统，很多操作都是自动完成的：

- 可以将你的文档分区到不同容器或者 分片 中，这些文档可能被存在一个节点或者多个节点。
- 跨节点平衡集群中节点间的索引与搜索负载。
- 自动复制你的数据以提供冗余副本，防止硬件错误导致数据丢失。
- 自动在节点之间路由，以帮助你找到你想要的数据。
- 无缝扩展或者恢复你的集群。

当你在阅读这本书时，你会发现到有关 Elasticsearch 的分布式特性分布式特性的补充章节。在这些章节中你会了解到如何扩展集群以及故障转移（《分布式集群》），如何处理文档存储（《分布式文档》），如何执行分布式搜索(《分布式搜索》）

这一部分不是必须要看的——你不懂它们也能正常使用 Elasticsearch。但是帮助你更加全面完整地了解 Elasticsearch。你也可以在之后需要的时候再回来翻阅它们。

总结

到目前为止，你应该已经知道 Elasticsearch 可以实现哪些功能，入门上手是非常简单的。只需要最少的知识和配置就可以开始使用 Elasticsearch 也是它的追求。学习 Elasticsearch 最好的方法就是开始使用它：进行的检索与搜索吧！

当然，学得越多，你的生产力就越高。你也就能对特定的内容进行微调，得到更适合你的结果。

之后的章节，我们将会引领你从新手晋级到专家。每一个章节都会解释一个要点，同时我们也会提供专家级别的小提示。如果你只是刚刚起步，这些提示可能并不是很适合你。Elasticsearch 会在一开始设置很多合理的默认值。你可以在需要提升性能的时候再重新回顾它们。

集群

补充章节

正如前文提到的，这就是第个补充的章节，这里会介绍 Elasticsearch 如何在分布式环境中运行。 本章解释了常用术语，比如 集群 *(cluster)*, 节点 *(node)* 以及 分片 *(shard)*，以及如何横向扩展主机，如何处理硬件故障。

尽管这一章不是必读章节 —— 你可以完全不用理会分片，复制以及故障恢复就能长时间使用 Elasticsearch。你可以先跳过这一章节，然后在你需要的时候再回来。

你可以随时根据你的需要扩展 Elasticsearch。你可以购买配置更好的主机 (*vertical scale* or *scaling up*) 或者购买更多的主机 (*horizontal scale* or *scaling out*) 来达到扩展的目的。

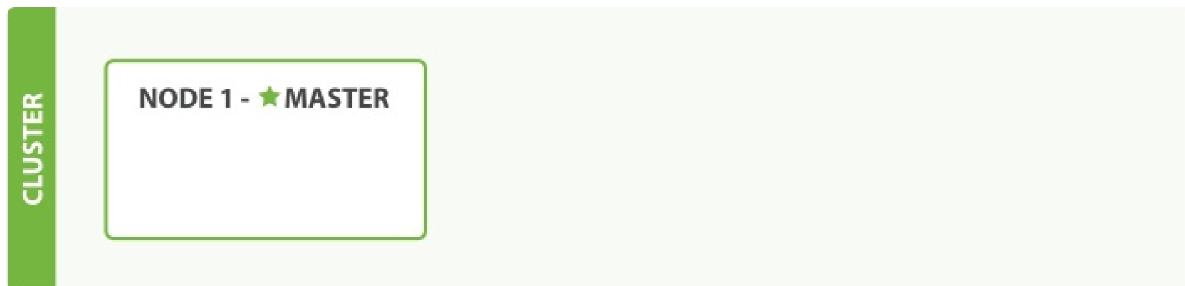
硬件越强大，Elasticsearch 运行的也就越快，但是垂直扩展 (vertical scale) 方式也有它的局限性。真正的扩展来自于横向扩展 (horizontal scale) 方式，在集群中添加更多的节点，这样能在节点之间分配负载。

对于大多数数据库来说，横向扩展意味着你的程序往往需要大改，以充分使用这些新添加的设备。相比而言，Elasticsearch 自带 分布式功能：他知道如何管理多个节点并提供高可用性。这也就意味着你的程序根本不需要为扩展做任何事情。

在这一章节，我们将要探索如何根据你的需要创建你的 集群，节点 以及 分片，并保障硬件故障后，你的数据依旧的安全。

空集群

如果我们启用一个既没有数据，也没有索引的单一节点，那我们的集群看起来就像是这样



节点 是 Elasticsearch 运行中的实例，而 集群 则包含一个或多个具有相同 `cluster.name` 的节点，它们协同工作，共享数据，并共同分担工作负荷。由于节点是从属集群的，集群会自我重组来均匀地分发数据。

集群中的一个节点会被选为 *master* 节点，它将负责管理集群范畴的变更，例如创建或删除索引，添加节点到集群或从集群删除节点。master 节点无需参与文档层面的变更和搜索，这意味着仅有一个 master 节点并不会因流量增长而成为瓶颈。任意一个节点都可以成为 master 节点。我们例举的集群只有一个节点，因此它会扮演 master 节点的角色。

作为用户，我们可以访问包括 master 节点在内的集群中的任一节点。每个节点都知道各个文档的位置，并能够将我们的请求直接转发到拥有我们想要的数据的节点。无论我们访问的是哪个节点，它都会控制从拥有数据的节点收集响应的过程，并返回给客户端最终的结果。这一切都是由 Elasticsearch 透明管理的。

集群健康

在 Elasticsearch 集群中可以监控统计很多信息，其中最重要的就是：集群健康(**cluster health**)。它的 `status` 有 `green` 、 `yellow` 、 `red` 三种；

```
GET /_cluster/health
```

在一个没有索引的空集群中，它将返回如下信息：

```
{
   "cluster_name":          "elasticsearch",
   "status":                "green", <1>
   "timed_out":             false,
   "number_of_nodes":       1,
   "number_of_data_nodes":  1,
   "active_primary_shards": 0,
   "active_shards":         0,
   "relocating_shards":     0,
   "initializing_shards":   0,
   "unassigned_shards":     0
}
```

1. `status` 是我们最应该关注的字段。

`status` 可以告诉我们当前集群是否处于一个可用的状态。三种颜色分别代表：

状态	意义
`green`	所有主分片和从分片都可用
`yellow`	所有主分片可用，但存在不可用的从分片
`red`	存在不可用的主要分片

在接下来的章节，我们将学习一下什么是主要分片(**primary shard**) 和 从分片(**replica shard**)，并说明这些状态在实际环境中的意义。

添加索引

为了将数据添加到 Elasticsearch，我们需要 索引(index) —— 存储关联数据的地方。实际上，索引只是一个 逻辑命名空间(logical namespace)，它指向一个或多个 分片(shards)。

分片(shard) 是 工作单元(worker unit) 底层的一员，它只负责保存索引中所有数据的一小片。在接下来的《深入分片》一章中，我们还将深入学习分片是如何运作的，但是现在你只要知道分片是一个独立的Lucene实例既可，并且它自身也是一个完整的搜索引擎。我们的文档存储并且被索引在分片中，但是我们的程序并不会直接与它们通信。取而代之，它们直接与索引进行通信的。

在 elasticsearch 中，分片用来分配集群中的数据。把分片想象成一个数据的容器。数据被存储在分片中，然后分片又被分配在集群的节点上。当你的集群扩展或者缩小时，elasticsearch 会自动的在节点之间迁移分配分片，以便集群保持均衡。

分片分为 主分片(primary shard) 以及 从分片(replica shard) 两种。在你的索引中，每一个文档都属于一个主分片，所以具体有多少主分片取决于你的索引能存储多少数据。

> 虽然理论上主分片对存储多少数据是没有限制的。分片的最大数量完全取决于你的实际状况：硬件的配置、文档的大小和复杂度、如何索引和查询你的文档，以及你期望的响应时间。

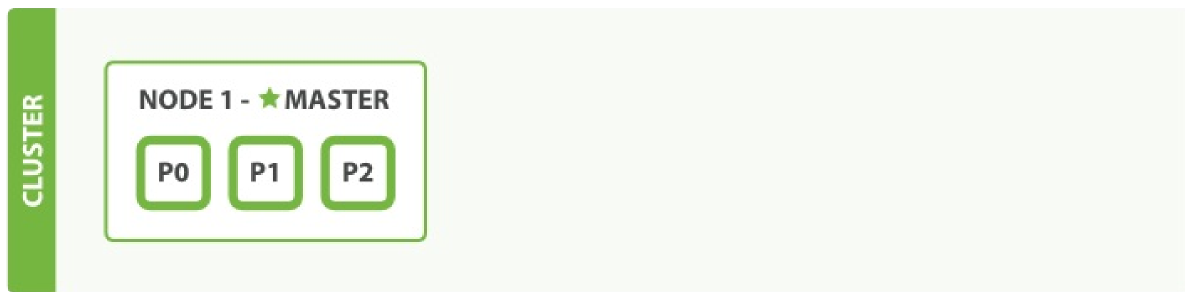
从分片只是主分片的一个副本，它用于提供数据的冗余副本，在硬件故障时提供数据保护，同时服务于搜索和检索这种只读请求。

索引中的主分片的数量在索引创建后就固定下来了，但是从分片的数量可以随时改变。

接下来，我们在空的单节点集群中上创建一个叫做 `blogs` 的索引。一个索引默认设置了5个主分片，但是为了演示，我们这里只设置3个主分片和一组从分片（每个主分片有一个从分片对应）：

```
PUT /blogs
{
   "settings" : {
      "number_of_shards" : 3,
      "number_of_replicas" : 1
   }
}
```

现在，我们的集群看起来就像下图所示了有索引的单节点集群，这三个主分片都被分配在 `Node 1` 。



如果我们现在查看 集群健康(**cluster-health**) ，我们将得到如下信息：

```
{
   "cluster_name":          "elasticsearch",
   "status":                "yellow", <1>
   "timed_out":             false,
   "number_of_nodes":       1,
   "number_of_data_nodes":  1,
   "active_primary_shards": 3,
   "active_shards":         3,
   "relocating_shards":     0,
   "initializing_shards":   0,
   "unassigned_shards":     3 <2>
}
```

1. 集群的 `status` 为 `yellow`.
2. 我们的三个从分片还没有被分配到节点上。

集群的健康状况 `yellow` 意味着所有的 主分片(**primary shards**) 启动并且运行了，这时集群已经可以成功的处理任意请求，但是 从分片(**replica shards**) 没有完全被激活。事实上，当前这三个从分片都处于 `unassigned` （未分配）的状态，它们还未被分配到节点上。在同一个节点上保存相同的数据副本是没有必要的，如果这个节点故障了，就等同于所有的数据副本也丢失了。

现在我们的集群已经可用了，但是依旧存在因硬件故障而导致数据丢失的风险。

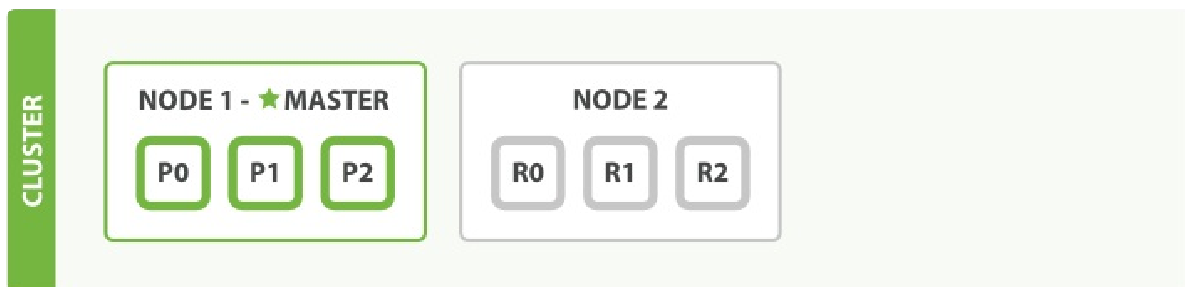
增加故障转移

在单一节点上运行意味着有单点故障的风险，没有数据冗余备份。幸运的是，我们可以启用另一个节点来保护我们的数据。

> ## 启动第二个节点
>
> 为了测试在增加第二个节点后发生了什么，你可以使用与第一个节点相同的方式启动第二个节点（你可以参考 入门-》安装-》运行 Elasticsearch 一章），而且在同一个目录——多个节点可以分享同一个目录。
>
> 只要第二个节点与第一个节点的 `cluster.name` 相同（参见 `./config/elasticsearch.yml` 文件中的配置），它就能自动发现并加入到第一个节点的集群中。如果没有，请结合日志找出问题所在。这可能是多播（multicast）被禁用，或者防火墙阻止了节点间的通信。

如果我们启动了第二个节点，这个集群应该叫做 双节点集群**(cluster-two-nodes)**

双节点集群——所有的主分片和从分片都被分配:



当第二个节点加入后，就产生了三个 从分片**(replica shards)** ，它们分别于三个主分片一一对应。也就意味着即使有一个节点发生了损坏，我们可以保证数据的完整性。

所有被索引的新文档都会先被存储在主分片中，之后才会被平行复制到关联的从分片上。这样可以确保我们的文档在主节点和从节点上都能被检索。

当前， `cluster-health` 的状态为 `green` ，这意味着所有的6个分片（三个主分片和三个从分片）都已激活：

```
{
   "cluster_name":          "elasticsearch",
   "status":                "green", <1>
   "timed_out":             false,
   "number_of_nodes":       2,
   "number_of_data_nodes":  2,
   "active_primary_shards": 3,
   "active_shards":         6,
   "relocating_shards":     0,
   "initializing_shards":   0,
   "unassigned_shards":     0
}
```

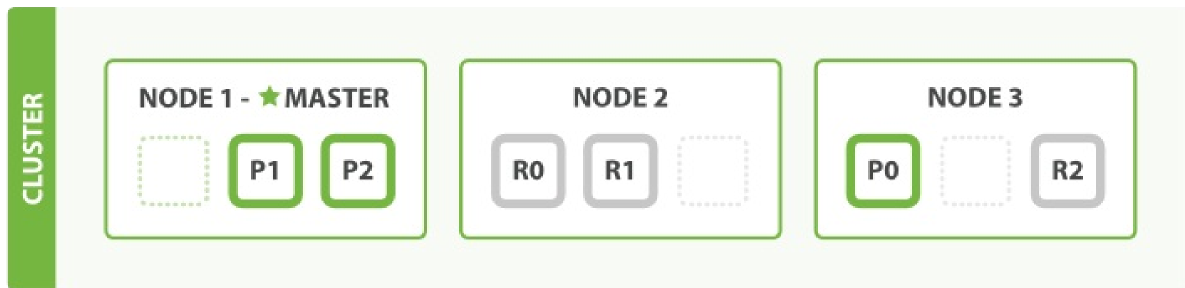
1. 集群的 `status` 是 `green` .

我们的集群不仅功能齐全的，并且具有高可用性。

横向扩展

随着应用需求的增长，我们该如何扩展？如果我们启动第三个节点，集群内会自动重组，这时便成为了三节点集群**(cluster-three-nodes)**

分片已经被重新分配以平衡负载：



在 `Node 1` 和 `Node 2` 中分别会有一个分片被移动到 `Node 3` 上，这样一来，每个节点上就都只有两个分片了。这意味着每个节点的硬件资源（CPU、RAM、I/O）被更少的分片共享，所以每个分片就会有更好的性能表现。

分片本身就是一个非常成熟的搜索引擎，它可以使用单个节点的所有资源。我们一共有6个分片（3个主分片和3个从分片），因此最多可以扩展到6个节点，每个节点上有一个分片，这样每个分片都可以使用到所在节点100%的资源了。

扩展更多

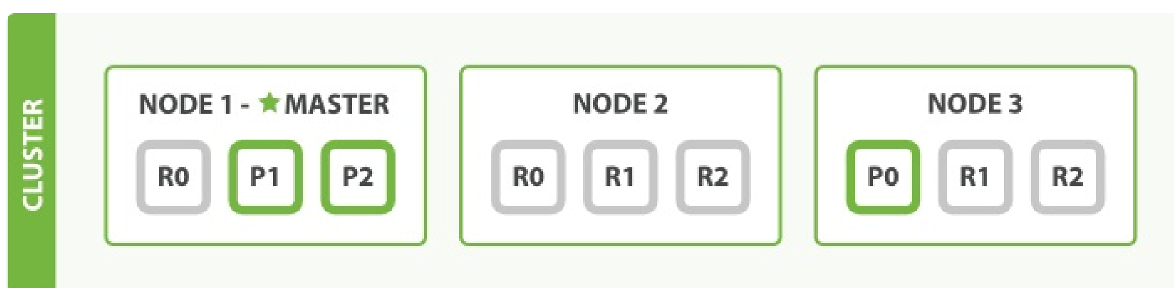
但是如果我们想要扩展到六个节点以上应该怎么办？

主分片的数量在索引创建的时候就已经指定了，实际上，这个数字定义了能存储到索引中的数据最大量（具体的数量取决于你的数据，硬件的使用情况）。例如，读请求——搜索或者文档恢复就可以由主分片或者从分片来执行，所以当你拥有更多份数据的时候，你就拥有了更大的吞吐量。

从分片的数量可以在运行的集群中动态的调整，这样我们就可以根据实际需求扩展或者缩小规模。接下来，我们来增加一下从分片组的数量：

```
PUT /blogs/_settings
{
   "number_of_replicas" : 2
}
```

增加 `number_of_replicas` 到2：

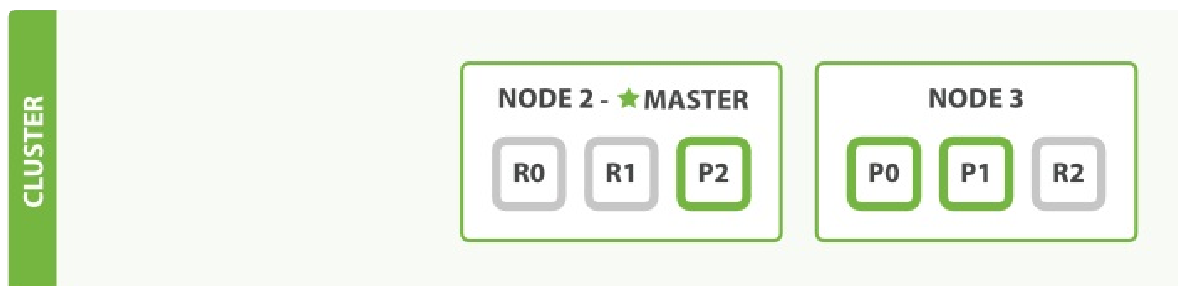


从图中可以看出，现在 `blogs` 的索引总共有9个分片：3个主分片和6个从分片。也就是说，现在我们就可以将总节点数扩展到9个，就又会变成一个节点一个分片的状态了。最终我们得到了三倍搜索性能的三节点集群。

> ## 提示
>
> 当然，仅仅是在同样数量的节点上增加从分片的数量是根本不能提高性能的，因为每个分片都有访问系统资源的权限。你需要升级硬件配置以提高吞吐量。
>
> 不过更多的从分片意味着我们有更多的冗余：通过上文的配置，我们可以承受两个节点的故障而不会丢失数据。

故障恢复

前文我们已经提到过 Elasticsearch 可以应对节点故障。让我们来尝试一下。如果我们把第一个节点杀掉，我们的集群就会如下图所示：



被杀掉的节点是主节点。而为了集群的正常工作必须需要一个主节点，所以首先进行的进程就是从各节点中选择了一个新的主节点： `Node 2` 。

主分片 `1` 和 `2` 在我们杀掉 `Node 1` 后就丢失了，我们的索引在丢失主节点的时候是不能正常工作的。如果我们在这个时候检查集群健康状态，将会显示 `red` ：存在不可用的主节点！

幸运的是，丢失的两个主分片的完整拷贝在存在于其他的节点上，所以新的主节点所完成的第一件事情就是将这些在 `Node 2` 和 `Node 3` 上的从分片提升为主分片，然后集群的健康状态就变回至 `yellow` 。这个提升的进程是瞬间完成了，就好像按了一下开关。

那么为什么集群健康状态依然是是 `yellow` 而不是 `green` 呢？是因为现在我们有3个主分片，但是我们之前设定了1个主分片有2个从分片，但是现在却只有1份从分片，所以状态无法变为 `green` ，不过我们可以不用太担心这里：当我们再次杀掉 `Node 2` 的时候，我们的程序依旧可以在没有丢失任何数据的情况下运行，因为 `Node 3` 中依旧拥有每个分片的备份。

如果我们重启 `Node 1` ，集群就能够重新分配丢失的从分片，这样结果就会与三节点两从集群一致。如果 `Node 1` 依旧还有旧节点的内容，系统会尝试重新利用他们，并只会复制在故障期间的变更数据。

到目前为止，我们已经清晰地了解了 Elasticsearch 的横向扩展以及数据安全的相关内容。接下来，我们将要继续讨论分片的生命周期等更多细节。

索引一个文档

文档通过 `索引` API被索引——存储并使其可搜索。但是最开始我们需要决定我们将文档存储在哪里。正如之前提到的，一篇文档通过 `_index` , `_type` 以及 `_id` 来确定它的唯一性。我们可以自己提供一个 `_id` ，或者也使用 `index` API 帮我们生成一个。

使用自己的ID

如果你的文档拥有天然的标示符（例如 `user_account` 字段或者文档中其他的标识值），这时你就可以提供你自己的 `_id` ，这样使用 `index` API：

```
PUT /{index}/{type}/{id}
{
  "field": "value",
  ...
}
```

几个例子。如果我们的索引叫做 `"website"` ，我们的类型叫做 `"blog"` ，然后我们选择 `"123"` 作为ID的编号。这时，请求就是这样的：

```
PUT /website/blog/123
{
  "title": "My first blog entry",
  "text":  "Just trying this out...",
  "date":  "2014/01/01"
}
```

Elasticsearch返回内容：

```
{
   "_index":    "website",
   "_type":     "blog",
   "_id":       "123",
   "_version":  1,
   "created":   true
}
```

这个返回值意味着我们的索引请求已经被成功创建，其中还包含了 `_index` , `_type` 以及 `_id` 的元数据，以及一个新的元素 `_version` 。

在Elasticsearch中，每一个文档都有一个版本号码。每当文档产生变化时（包括删除），`_version` 就会增大。在《版本控制》中，我们将会详细讲解如何使用 `_version` 的数字来确认你的程序不会随意替换掉不想覆盖的数据。

自增ID

如果我们的数据中没有天然的标示符，我们可以让Elasticsearch为我们自动生成一个。请求的结构发生了变化：我们把 `PUT` ——“把文档存储在这个地址中”变量变成了 `POST` ——“把文档存储在这个地址下”。

这样一来，请求中就只包含 `_index` 和 `_type` 了：

```
POST /website/blog/
{
  "title": "My second blog entry",
  "text":  "Still trying this out...",
  "date":  "2014/01/01"
}
```

这次的反馈和之前基本一样，只有 `_id` 改成了系统生成的自增值:

```
{
   "_index":    "website",
   "_type":     "blog",
   "_id":       "wM0OSFhDQXGZAWDf0-drSA",
   "_version":  1,
   "created":   true
}
```

自生成ID是由22个字母组成的，安全 *universally unique identifiers* 或者被称为UUIDs。

文档是什么？

在很多程序中，大部分实体或者对象都被序列化为包含键和值的JSON对象。键是一个字段或者属性的名字，值可以是一个字符串、数字、布尔值、对象、数组或者是其他的特殊类型，比如代表日期的字符串或者代表地理位置的对象：

```
{
    "name":         "John Smith",
    "age":          42,
    "confirmed":    true,
    "join_date":    "2014-06-01",
    "home": {
        "lat":      51.5,
        "lon":      0.1
    },
    "accounts": [
        {
            "type": "facebook",
            "id":   "johnsmith"
        },
        {
            "type": "twitter",
            "id":   "johnsmith"
        }
    ]
}
```

通常情况下，我们使用可以互换对象和文档。然而，还是有一个区别的。对象(object)仅仅是一个JSON对象,类似于哈希，哈希映射，字典或关联数组。对象(Objects)则可以包含其他对象(Objects)。

在Elasticsearch中，文档这个单词有特殊的含义。它指的是在Elasticsearch中被存储到唯一ID下的由最高级或者根对象 *(root object)*序列化而来的JSON。

文档元数据

一个文档不只包含了数据。它还包含了元数据*(metadata)* —— 关于文档的信息。有三个元数据元素是必须存在的，它们是：

名字	说明
`_index`	文档存储的地方
`_type`	文档代表的对象种类
`_id`	文档的唯一编号

`_index`

索引 类似于传统数据库中的"数据库"——也就是我们存储并且索引相关数据的地方。

> **TIP**：

在Elasticsearch中，我们的数据都在分片中被存储以及索引，索引只是一个逻辑命名空间，它可以将一个或多个分片组合在一起。然而，这只是一个内部的运作原理——我们的程序可以根本不用关心分片。对于我们的程序来说，我们的文档存储在索引中。剩下的交给Elasticsearch就可以了。

我们将会在《索引管理》章节中探讨如何创建并管理索引。但是现在，我们只需要让Elasticsearch帮助我们创建索引。我们只需要选择一个索引的名字。这个名称必须要全部小写，也不能以下划线开头，不能包含逗号。我们可以用 `website` 作为我们索引的名字。

`_type`

在程序中，我们使用对象代表“物品”，比如一个用户、一篇博文、一条留言或者一个邮件。每一个对象都属于一种类型，类型定义了对象的属性或者与数据的关联。用户类的对象可能就会包含名字、性别、年龄以及邮箱地址等。

在传统的数据库中，我们总是将同类的数据存储在同一个表中，因为它们的数据格式是相同的。同理，在Elasticsearch中，我们使用同样类型的文档来代表同类“事物”，也是因为它们的数据结构是相同的。

每一个类型都拥有自己的映射(mapping)或者结构定义，它们定义了当前类型下的数据结构，类似于数据库表中的列。所有类型下的文档会被存储在同一个索引下，但是映射会告诉Elasticsearch不同的数据应该如何被索引。

我们将会在《映射》中探讨如何制定或者管理映射，但是目前为止，我们只需要依靠Elasticsearch来自动处理数据结构。

`_id`

*id*是一个字符串，当它与 `_index` 以及 `_type` 组合时，就可以来代表Elasticsearch中一个特定的文档。我们创建了一个新的文档时，你可以自己提供一个 `_id` ，或者也可以让Elasticsearch帮你生成一个。

其他元数据

在文档中还有一些其他的元数据，我们将会在《映射》章节中详细讲解。使用上面罗列的元素，我们已经可以在Elasticsearch中存储文档或者通过ID来搜索已经保存的文档了。

搜索文档

要从Elasticsearch中获取文档，我们需要使用同样的 `_index` ， `_type` 以及 `_id` 但是不同的HTTP变量 `GET` ：

```
GET /website/blog/123?pretty
```

返回结果包含了之前提到的内容，以及一个新的字段 `_source` ，它包含我们在最初创建索引时的原始JSON文档。

```
{
  "_index" :   "website",
  "_type" :    "blog",
  "_id" :      "123",
  "_version" : 1,
  "found" :    true,
  "_source" :  {
      "title": "My first blog entry",
      "text":  "Just trying this out..."
      "date":  "2014/01/01"
  }
}
```

> ## pretty
>
> 在任意的查询字符串中添加 `pretty` 参数，类似上面的请求，Elasticsearch就可以得到优美打印的更加易于识别的JSON结果。 `_source` 字段不会执行优美打印，它的样子取决于我们录入的样子。

GET请求的返回结果中包含 `{"found": true}` 。这意味着这篇文档确实被找到了。如果我们请求了一个不存在的文档，我们依然会得到JSON反馈，只是 `found` 的值会变为 `false` 。

同样，HTTP返回码也会由 `'200 OK'` 变为 `'404 Not Found'` 。我们可以在 `curl` 后添加 `-i` ，这样你就能得到反馈头文件：

```
curl -i -XGET /website/blog/124?pretty
```

反馈结果就会是这个样子：

```
HTTP/1.1 404 Not Found
Content-Type: application/json; charset=UTF-8
Content-Length: 83

{
  "_index" : "website",
  "_type" :  "blog",
  "_id" :    "124",
  "found" :  false
}
```

检索文档中的一部分

通常， `GET` 请求会将整个文档放入 `_source` 字段中一并返回。但是可能你只需要 `title` 字段。你可以使用 `_source` 得到指定字段。如果需要多个字段你可以使用逗号分隔：

```
GET /website/blog/123?_source=title,text
```

现在 `_source` 字段中就只会显示你指定的字段：

```
{
  "_index" :   "website",
  "_type" :    "blog",
  "_id" :      "123",
  "_version" : 1,
  "exists" :   true,
  "_source" : {
      "title": "My first blog entry" ,
      "text":  "Just trying this out..."
  }
}
```

或者你只想得到 `_source` 字段而不要其他的元数据，你可以这样请求：

```
GET /website/blog/123/_source
```

这样结果就只返回:

```
{
   "title": "My first blog entry",
   "text":  "Just trying this out...",
   "date":  "2014/01/01"
}
```

检查文档是否存在

如果确实想检查一下文档是否存在，你可以试用 `HEAD` 来替代 `GET` 方法，这样就是会返回HTTP头文件：

```
curl -i -XHEAD /website/blog/123
```

如果文档存在，Elasticsearch将会返回 `200 OK` 的状态码：

```
HTTP/1.1 200 OK
Content-Type: text/plain; charset=UTF-8
Content-Length: 0
```

如果不存在将会返回 `404 Not Found` 状态码：

```
curl -i -XHEAD /website/blog/124
```

```
HTTP/1.1 404 Not Found
Content-Type: text/plain; charset=UTF-8
Content-Length: 0
```

当然，这个反馈只代表了你查询的那一刻文档不存在，但是不代表几毫秒后它不存在，很可能与此同时，另一个进程正在创建文档。

更新整个文档

在Documents中的文档是不可改变的。所以如果我们需要改变已经存在的文档，我们可以使用《索引》中提到的 `index` API来重新索引或者替换掉它：

```
PUT /website/blog/123
{
  "title": "My first blog entry",
  "text":  "I am starting to get the hang of this...",
  "date":  "2014/01/02"
}
```

在反馈中，我们可以发现Elasticsearch已经将 `_version` 数值增加了：

```
{
  "_index" :   "website",
  "_type" :    "blog",
  "_id" :      "123",
  "_version" : 2,
  "created":   false <1>
}
```

1. `created` 被标记为 `false` 是因为在同索引、同类型下已经存在同ID的文档。

在内部，Elasticsearch已经将旧文档标记为删除并且添加了新的文档。旧的文档并不会立即消失，但是你也无法访问他。Elasticsearch会在你继续添加更多数据的时候在后台清理已经删除的文件。

在本章的后面，我们将会在《局部更新》中介绍最新更新的API。这个API允许你修改局部，但是原理和下方的完全一样：

1. 从旧的文档中检索JSON
2. 修改它
3. 删除修的文档
4. 索引一个新的文档

唯一不同的是，使用了 `update` API你就不需要使用 `get` 然后再操作 `index` 请求了。

创建一个文档

当我们索引一个文档时，如何确定我们是创建了一个新的文档还是覆盖了一个已经存在的文档呢？

请牢记 `_index` , `_type` 以及 `_id` 组成了唯一的文档标记，所以为了确定我们创建的是全新的内容，最简单的方法就是使用 `POST` 方法，让Elasticsearch自动创建不同的 `_id` ：

```
POST /website/blog/
{ ... }
```

然而，我们可能已经决定好了 `_id` ，所以需要告诉Elasticsearch只有当 `_index` ， `_type` 以及 `_id` 这3个属性全部相同的文档不存在时才接受我们的请求。实现这个目的有两种方法，他们实质上是一样的，你可以选择你认为方便的那种：

第一种是在查询中添加 `op_type` 参数：

```
PUT /website/blog/123?op_type=create
{ ... }
```

或者在请求最后添加 `/_create` ：

```
PUT /website/blog/123/_create
{ ... }
```

如果成功创建了新的文档，Elasticsearch将会返回常见的元数据以及 `201 Created` 的HTTP反馈码。

而如果存在同名文件，Elasticsearch将会返回一个 `409 Conflict` 的HTTP反馈码，以及如下方的错误信息：

```
{
  "error" : "DocumentAlreadyExistsException[[website][4] [blog][123]:
             document already exists]",
  "status" : 409
}
```

删除一个文档

删除文档的基本模式和之前的基本一样，只不过是需要更换成 `DELETE` 方法：

```
DELETE /website/blog/123
```

如果文档存在，那么Elasticsearch就会返回一个 `200 OK` 的HTTP响应码，返回的结果就会像下面展示的一样。请注意 `_version` 的数字已经增加了。

```
{
  "found" :    true,
  "_index" :   "website",
  "_type" :    "blog",
  "_id" :      "123",
  "_version" : 3
}
```

如果文档不存在，那么我们就会得到一个 `404 Not Found` 的响应码，返回的内容就会是这样的：

```
{
  "found" :    false,
  "_index" :   "website",
  "_type" :    "blog",
  "_id" :      "123",
  "_version" : 4
}
```

尽管文档并不存在（ `"found"` 值为 `false` ），但是 `_version` 的数值仍然增加了。这个就是内部管理的一部分，它保证了我们在多个节点间的不同操作的顺序都被正确标记了。

正如我在《更新》一章中提到的，删除一个文档也不会立即生效，它只是被标记成已删除。Elasticsearch将会在你之后添加更多索引的时候才会在后台进行删除内容的清理。

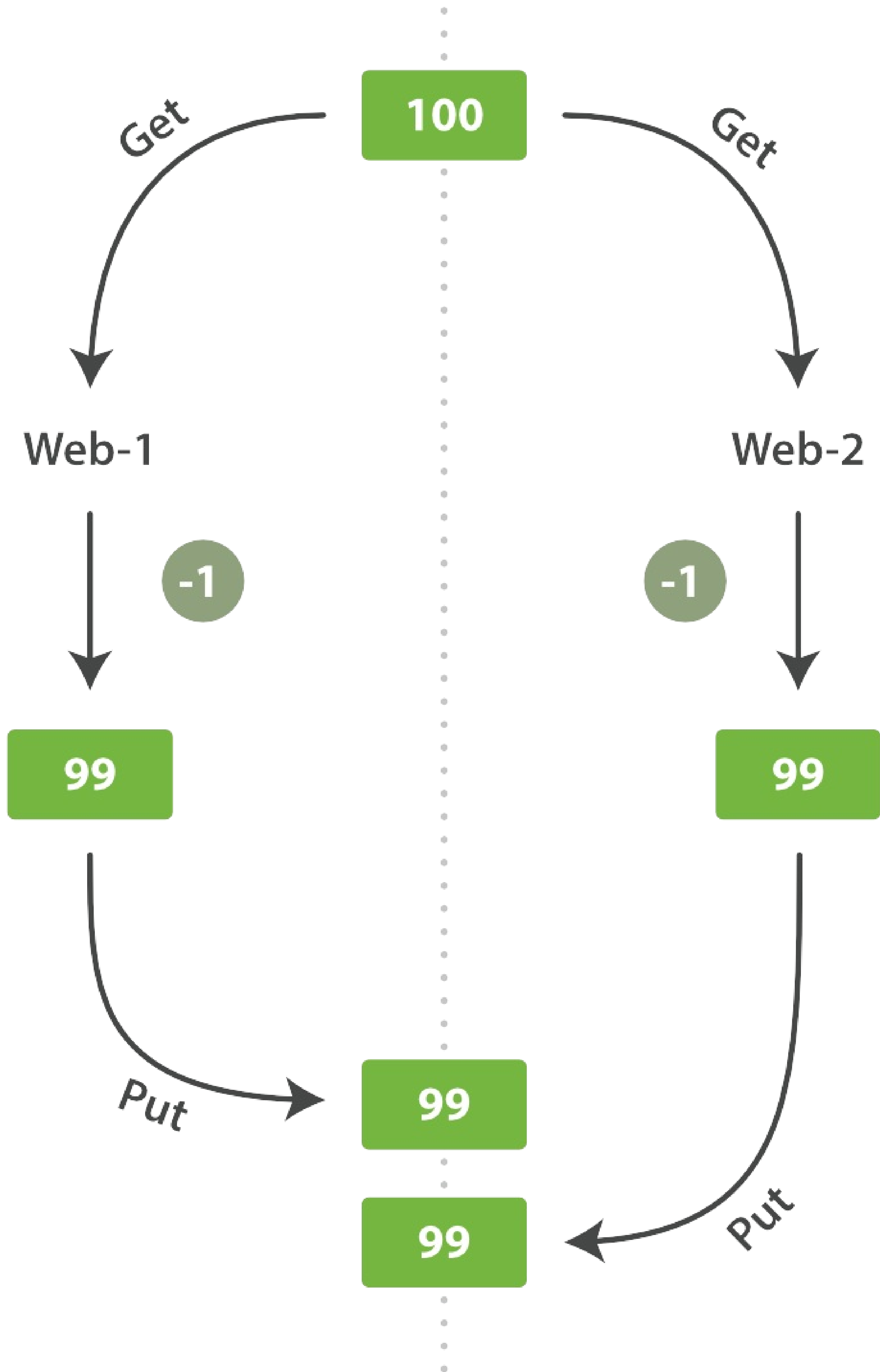
处理冲突

当你使用 `索引` API来更新一个文档时，我们先看到了原始文档，然后修改它，最后一次性地将整个新文档进行再次索引处理。Elasticsearch会根据请求发出的顺序来选择出最新的一个文档进行保存。但是，如果在你修改文档的同时其他人也发出了指令，那么他们的修改将会丢失。

很长时间以来，这其实都不是什么大问题。或许我们的主要数据还是存储在一个关系数据库中，而我们只是将为了可以搜索，才将这些数据拷贝到Elasticsearch中。或许发生多个人同时修改一个文件的概率很小，又或者这些偶然的数据丢失并不会影响到我们的正常使用。

但是有些时候如果我们丢失了数据就会出大问题。想象一下，如果我们使用Elasticsearch来存储一个网店的商品数量。每当我们卖出一件，我们就会将这个数量减少一个。

突然有一天，老板决定来个大促销。瞬间，每秒就产生了多笔交易。并行处理，多个进程来处理交易：


100
Get
Get
Web-1
Web-2
-1
-1
99
99
Put
99
99
Put

`web_1` 中 `库存量` 的变化丢失的原因是 `web_2` 并不知道它所得到的 `库存量` 数据是是过期的。这样就会导致我们误认为还有很多货存，最终顾客就会对我们的行为感到失望。

当我们对数据修改得越频繁，或者在读取和更新数据间有越长的空闲时间，我们就越容易丢失掉我们的数据。

以下是两种能避免在并发更新时丢失数据的方法：

悲观并发控制（PCC）

这一点在关系数据库中被广泛使用。假设这种情况很容易发生，我们就可以阻止对这一资源的访问。典型的例子就是当我们在读取一个数据前先锁定这一行，然后确保只有读取到数据的这个线程可以修改这一行数据。

乐观并发控制（OCC）

Elasticsearch所使用的。假设这种情况并不会经常发生，也不会去阻止某一数据的访问。然而，如果基础数据在我们读取和写入的间隔中发生了变化，更新就会失败。这时候就由程序来决定如何处理这个冲突。例如，它可以重新读取新数据来进行更新，又或者它可以将这一情况直接反馈给用户。

乐观并发控制

Elasticsearch是分布式的。当文档被创建、更新或者删除时，新版本的文档就会被复制到集群中的其他节点上。Elasticsearch即是同步的又是异步的，也就是说复制的请求被平行发送出去，然后可能会混乱地到达目的地。这就需要一种方法能够保证新的数据不会被旧数据所覆盖。

我们在上文提到每当有 `索引` 、 `put` 和 `删除` 的操作时，无论文档有没有变化，它的 `_version` 都会增加。Elasticsearch使用 `_version` 来确保所有的改变操作都被正确排序。如果一个旧的版本出现在新版本之后，它就会被忽略掉。

我们可以利用 `_version` 的优点来确保我们程序修改的数据冲突不会造成数据丢失。我们可以按照我们的想法来指定 `_version` 的数字。如果数字错误，请求就是失败。

我们来创建一个新的博文:

```
PUT /website/blog/1/_create
{
  "title": "My first blog entry",
  "text":  "Just trying this out..."
}
```

反馈告诉我们这是一个新建的文档，它的 `_version` 是 `1` 。假设我们要编辑它，把这个数据加载到网页表单中，修改完毕然后保存新版本。

首先我们先要得到文档：

```
GET /website/blog/1
```

返回结果显示 `_version` 为 `1` ：

```
{
  "_index" :   "website",
  "_type" :    "blog",
  "_id" :      "1",
  "_version" : 1,
  "found" :    true,
  "_source" :  {
      "title": "My first blog entry",
      "text":  "Just trying this out..."
  }
}
```

现在，我们试着重新索引文档以保存变化，我们这样指定了 `version` 的数字：

```
PUT /website/blog/1?version=1 <1>
{
  "title": "My first blog entry",
  "text":  "Starting to get the hang of this..."
}
```

1. 我们只希望当索引中文档的 `_version` 是 `1` 时，更新才生效。

请求成功相应，返回内容告诉我们 `_version` 已经变成了 `2` ：

```
{
  "_index":   "website",
  "_type":    "blog",
  "_id":      "1",
  "_version": 2
  "created":  false
}
```

然而，当我们再执行同样的索引请求，并依旧指定 `version=1` 时，Elasticsearch就会返回一个 `409 Conflict` 的响应码，返回内容如下：

```
{
  "error" : "VersionConflictEngineException[[website][2] [blog][1]:
             version conflict, current [2], provided [1]]",
  "status" : 409
}
```

这里面指出了文档当前的 `_version` 数字是 `2` ，而我们要求的数字是 `1` 。

我们需要做什么取决于我们程序的需求。比如我们可以告知用户已经有其它人修改了这个文档，你应该再保存之前看一下变化。而对于上文提到的 `库存量` 问题，我们可能需要重新读取一下最新的文档，然后显示新的数据。

所有的有关于更新或者删除文档的API都支持 `version` 这个参数，有了它你就通过修改你的程序来使用乐观并发控制。

使用外部系统的版本

还有一种常见的情况就是我们还是使用其他的数据库来存储数据，而Elasticsearch只是帮我们检索数据。这也就意味着主数据库只要发生的变更，就需要将其拷贝到Elasticsearch中。如果多个进程同时发生，就会产生上文提到的那些并发问题。

如果你的数据库已经存在了版本号码，或者也可以代表版本的 `时间戳` 。这是你就可以在Elasticsearch的查询字符串后面添加 `version_type=external` 来使用这些号码。版本号码必须要是大于零小于 `9.2e+18` （Java中long的最大正值）的整数。

Elasticsearch在处理外部版本号时会与对内部版本号的处理有些不同。它不再是检查 `_version` 是否与请求中指定的数值相同,而是检查当前的 `_version` 是否比指定的数值小。如果请求成功，那么外部的版本号就会被存储到文档中的 `_version` 中。

外部版本号不仅可以在索引和删除请求时使用，还可以在创建时使用。

例如，创建一篇使用外部版本号为 `5` 的博文，我们可以这样操作：

```
PUT /website/blog/2?version=5&version_type=external
{
  "title": "My first external blog entry",
  "text":  "Starting to get the hang of this..."
}
```

在返回结果中，我们可以发现 `_version` 是 `5` ：

```
{
  "_index":   "website",
  "_type":    "blog",
  "_id":      "2",
  "_version": 5,
  "created":  true
}
```

现在我们更新这个文档，并指定 `version` 为 `10` ：

```
PUT /website/blog/2?version=10&version_type=external
{
  "title": "My first external blog entry",
  "text":  "This is a piece of cake..."
}
```

请求被成功执行并且 `version` 也变成了 `10` ：

```
{
  "_index":   "website",
  "_type":    "blog",
  "_id":      "2",
  "_version": 10,
  "created":  false
}
```

如果你再次执行这个命令，你会得到之前的错误提示信息，因为你所指定的版本号并没有大于当前Elasticsearch中的版本号。

更新文档中的一部分

在《更新》一章中，我们讲到了要是想更新一个文档，那么就需要去取回数据，更改数据然后将整个文档进行重新索引。当然，你还可以通过使用`更新`API来做部分更新，比如增加一个计数器。

正如我们提到的，文档不能被修改，它们只能被替换掉。`更新`API也必须遵循这一法则。从表面看来，貌似是文档被替换了。对内而言，它必须按照找回-修改-索引的流程来进行操作与管理。不同之处在于这个流程是在一个片(shard) 中完成的，因此可以节省多个请求所带来的网络开销。除了节省了步骤，同时我们也能减少多个进程造成冲突的可能性。

使用`更新`请求最简单的一种用途就是添加新数据。新的数据会被合并到现有数据中，而如果存在相同的字段，就会被新的数据所替换。例如我们可以为我们的博客添加`tags`和`views`字段：

```
POST /website/blog/1/_update
{
   "doc" : {
      "tags" : [ "testing" ],
      "views": 0
   }
}
```

如果请求成功，我们就会收到一个类似于`索引`时返回的内容:

```
{
   "_index" :   "website",
   "_id" :      "1",
   "_type" :    "blog",
   "_version" : 3
}
```

再次取回数据，你可以在`_source`中看到更新的结果：

```
{
   "_index":    "website",
   "_type":     "blog",
   "_id":       "1",
   "_version":  3,
   "found":     true,
   "_source": {
      "title":  "My first blog entry",
      "text":   "Starting to get the hang of this...",
      "tags": [ "testing" ], <1>
      "views":  0 <1>
   }
}
```

1. 新的数据已经添加到了字段 `_source` 中。

使用脚本进行更新

我们将会在《脚本》一章中学习更详细的内容，我们现在只需要了解一些在Elasticsearch中使用API无法直接完成的自定义行为。默认的脚本语言叫做MVEL，但是Elasticsearch也支持JavaScript, Groovy 以及 Python。

MVEL是一个简单高效的JAVA基础动态脚本语言，它的语法类似于Javascript。你可以在Elasticsearch scripting docs 以及 MVEL website了解更多关于MVEL的信息。

脚本语言可以在 `更新` API中被用来修改 `_source` 中的内容，而它在脚本中被称为 `ctx._source` 。例如，我们可以使用脚本来增加博文中 `views` 的数字：

```
POST /website/blog/1/_update
{
   "script" : "ctx._source.views+=1"
}
```

我们同样可以使用脚本在 `tags` 数组中添加新的tag。在这个例子中，我们把新的tag声明为一个变量，而不是将他写死在脚本中。这样Elasticsearch就可以重新使用这个脚本进行tag的添加，而不用再次重新编写脚本了：

```
POST /website/blog/1/_update
{
   "script" : "ctx._source.tags+=new_tag",
   "params" : {
      "new_tag" : "search"
   }
}
```

获取文档，后两项发生了变化：

```
{
   "_index":    "website",
   "_type":     "blog",
   "_id":       "1",
   "_version":  5,
   "found":     true,
   "_source": {
      "title":  "My first blog entry",
      "text":   "Starting to get the hang of this...",
      "tags":  ["testing", "search"], <1>
      "views":  1 <2>
   }
}
```

1. `tags` 数组中出现了 `search` 。
2. `views` 字段增加了。

我们甚至可以使用 `ctx.op` 来根据内容选择是否删除一个文档：

```
POST /website/blog/1/_update
{
   "script" : "ctx.op = ctx._source.views == count ? 'delete' : 'none'",
    "params" : {
        "count": 1
    }
}
```

更新一篇可能不存在的文档

想象一下，我们可能需要在Elasticsearch中存储一个页面计数器。每次用户访问这个页面，我们就增加一下当前页面的计数器。但是如果这是个新的页面，我们不能确保这个计数器已经存在。如果我们试着去更新一个不存在的文档，更新操作就会失败。

为了防止上述情况的发生，我们可以使用 `upsert` 参数来设定文档不存在时，它应该被创建：

```
POST /website/pageviews/1/_update
{
   "script" : "ctx._source.views+=1",
   "upsert": {
       "views": 1
   }
}
```

首次运行这个请求时， `upsert` 的内容会被索引成新的文档，它将 `views` 字段初始化为 `1` 。当之后再请求时，文档已经存在，所以 `脚本` 更新就会被执行， `views` 计数器就会增加。

更新和冲突

在本节的开篇我们提到了当取回与重新索引两个步骤间的时间越少，发生改变冲突的可能性就越小。但它并不能被完全消除，在 `更新` 的过程中还可能存在另一个进程进行重新索引的可能性。

为了避免丢失数据， `更新` API会在获取步骤中获取当前文档中的 `_version` ，然后将其传递给重新索引步骤中的 `索引` 请求。如果其他的进程在这两部之间修改了这个文档，那么 `_version` 就会不同，这样更新就会失败。

对于很多的局部更新来说，文档有没有发生变化实际上是不重要的。例如，两个进程都要增加页面浏览的计数器，谁先谁后其实并不重要 —— 发生冲突时只需要重新来过即可。

你可以通过设定 `retry_on_conflict` 参数来设置自动完成这项请求的次数，它的默认值是 `0` 。

```
POST /website/pageviews/1/_update?retry_on_conflict=5 <1>
{
   "script" : "ctx._source.views+=1",
   "upsert": {
       "views": 0
   }
}
```

1. 失败前重新尝试5次

这个参数非常适用于类似于增加计数器这种无关顺序的请求，但是还有些情况的顺序就是很重要的。例如上一节提到的情况，你可以参考乐观并发控制以及悲观并发控制来设定文档的版本号。

获取多个文档

尽管Elasticsearch已经很快了，但是它依旧可以更快。你可以将多个请求合并到一个请求中以节省网络开销。如果你需要从Elasticsearch中获取多个文档，你可以使用*multi-get* 或者 `mget` API来取代一篇又一篇文档的获取。

`mget` API需要一个 `docs` 数组，每一个元素包含你想要的文档的 `_index`，`_type` 以及 `_id`。你也可以指定 `_source` 参数来设定你所需要的字段：

```
GET /_mget
{
   "docs" : [
      {
         "_index" : "website",
         "_type" :  "blog",
         "_id" :    2
      },
      {
         "_index" : "website",
         "_type" :  "pageviews",
         "_id" :    1,
         "_source": "views"
      }
   ]
}
```

返回值包含了一个 `docs` 数组，这个数组以请求中指定的顺序每个文档包含一个响应。每一个响应都和独立的 `get` 请求返回的响应相同：

```
{
   "docs" : [
      {
         "_index" :   "website",
         "_id" :      "2",
         "_type" :    "blog",
         "found" :    true,
         "_source" : {
            "text" :  "This is a piece of cake...",
            "title" : "My first external blog entry"
         },
         "_version" : 10
      },
      {
         "_index" :   "website",
         "_id" :      "1",
         "_type" :    "pageviews",
         "found" :    true,
         "_version" : 2,
         "_source" : {
            "views" : 2
         }
      }
   ]
}
```

如果你所需要的文档都在同一个 `_index` 或者同一个 `_type` 中，你就可以在URL中指定一个默认的 `/_index` 或是 `/_index/_type` 。

你也可以在单独的请求中重写这个参数：

```
GET /website/blog/_mget
{
   "docs" : [
      { "_id" : 2 },
      { "_type" : "pageviews", "_id" :   1 }
   ]
}
```

事实上，如果所有的文档拥有相同的 `_index` 以及 `_type` ，直接在请求中添加 `ids` 的数组即可：

```
GET /website/blog/_mget
{
   "ids" : [ "2", "1" ]
}
```

请注意，我们所请求的第二篇文档不存在，这是就会返回如下内容：

```
{
  "docs" : [
    {
      "_index" :   "website",
      "_type" :    "blog",
      "_id" :      "2",
      "_version" : 10,
      "found" :    true,
      "_source" : {
        "title":   "My first external blog entry",
        "text":    "This is a piece of cake..."
      }
    },
    {
      "_index" :   "website",
      "_type" :    "blog",
      "_id" :      "1",
      "found" :    false  <1>
    }
  ]
}
```

1. 文档没有被找到。

当第二篇文档没有被找到的时候也不会影响到其它文档的获取结果。每一个文档都会被独立展示。

注意：上方请求的HTTP状态码依旧是 `200` ，尽管有个文档没有找到。事实上，即使所有的文档都没有被找到，响应码也依旧是 `200` 。这是因为 `mget` 这个请求本身已经成功完成。要确定独立的文档是否被成功找到，你需要检查 `found` 标识。

批量更高效

与 `mget` 能同时允许帮助我们获取多个文档相同， `bulk` API可以帮助我们同时完成执行多个请求，比如： `create` ， `index` , `update` 以及 `delete` 。当你在处理类似于log等海量数据的时候，你就可以一下处理成百上千的请求，这个操作将会极大提高效率。

`bulk` 的请求主体的格式稍微有些不同：

```
{ action: { metadata }}\n
{ request body        }\n
{ action: { metadata }}\n
{ request body        }\n
...
```

这种格式就类似于一个用 `"\n"` 字符来连接的单行json一样。下面是两点注意事项：

- 每一行都结尾处都必须有换行字符 `"\n"` ，最后一行也要有。这些标记可以有效地分隔每行。
- 这些行里不能包含非转义字符，以免干扰数据的分析 —— 这也意味着JSON不能是pretty-printed样式。

TIP

在《bulk格式》一章中，我们将解释为何 `bulk` API要使用这种格式。

---

action/metadata 行指定了将要在哪个文档中执行什么操作。

其中*action*必须是 `index` , `create` , `update` 或者 `delete` 。*metadata* 需要指明需要被操作文档的 `_index` , `_type` 以及 `_id` ，例如删除命令就可以这样填写：

```
{ "delete": { "_index": "website", "_type": "blog", "_id": "123" }}
```

在你进行 `index` 以及 `create` 操作时，*request body* 行必须要包含文档的 `_source` 数据——也就是文档的所有内容。

同样，在执行 `update` API: `doc` , `upsert` , `script` 的时候，也需要包含相关数据。而在删除的时候就不需要*request body*行。

```
{ "create":  { "_index": "website", "_type": "blog", "_id": "123" }}
{ "title":    "My first blog post" }
```

如果没有指定 `_id` ，那么系统就会自动生成一个ID：

```
{ "index": { "_index": "website", "_type": "blog" }}
{ "title":    "My second blog post" }
```

完成以上所有请求的 `bulk` 如下：

```
POST /_bulk
{ "delete": { "_index": "website", "_type": "blog", "_id": "123" }} <1>
{ "create": { "_index": "website", "_type": "blog", "_id": "123" }}
{ "title":    "My first blog post" }
{ "index":  { "_index": "website", "_type": "blog" }}
{ "title":    "My second blog post" }
{ "update": { "_index": "website", "_type": "blog", "_id": "123", "_retry_on_conflict"
 : 3} }
{ "doc" : {"title" : "My updated blog post"} } <2>
```

1. 注意 `delete` 操作是如何处理*request body*的,你可以在它之后直接执行新的操作。
2. 请记住最后有换行符

Elasticsearch会返回含有 `items` 的列表、它的顺序和我们请求的顺序是相同的：

```
{
   "took": 4,
   "errors": false, <1>
   "items": [
      {  "delete": {
            "_index":   "website",
            "_type":    "blog",
            "_id":      "123",
            "_version": 2,
            "status":   200,
            "found":    true
      }},
      {  "create": {
            "_index":   "website",
            "_type":    "blog",
            "_id":      "123",
            "_version": 3,
            "status":   201
      }},
      {  "create": {
            "_index":   "website",
            "_type":    "blog",
            "_id":      "EiwfApScQiiy7TIKFxRCTw",
            "_version": 1,
            "status":   201
      }},
      {  "update": {
            "_index":   "website",
            "_type":    "blog",
            "_id":      "123",
            "_version": 4,
            "status":   200
      }}
   ]
}}
```

1. 所有的请求都被成功执行。

每一个子请求都会被单独执行，所以一旦有一个子请求失败了，并不会影响到其他请求的成功执行。如果一旦出现失败的请求，`error` 就会变为 `true` ，详细的错误信息也会出现在返回内容的下方：

```
POST /_bulk
{ "create": { "_index": "website", "_type": "blog", "_id": "123" }}
{ "title":    "Cannot create - it already exists" }
{ "index":  { "_index": "website", "_type": "blog", "_id": "123" }}
{ "title":    "But we can update it" }
```

请求中的 `create` 操作失败，因为 `123` 已经存在，但是之后针对文档 `123` 的 `index` 操作依旧被成功执行：

```
{
   "took": 3,
   "errors": true, <1>
   "items": [
      {  "create": {
            "_index":   "website",
            "_type":    "blog",
            "_id":      "123",
            "status":   409, <2>
            "error":    "DocumentAlreadyExistsException <3>
                        [[website][4] [blog][123]:
                        document already exists]"
      }},
      {  "index": {
            "_index":   "website",
            "_type":    "blog",
            "_id":      "123",
            "_version": 5,
            "status":   200 <4>
      }}
   ]
}
```

1. 至少有一个请求错误发生。
2. 这条请求的状态码为 `409 CONFLICT` 。
3. 错误信息解释了导致错误的原因。
4. 第二条请求的状态码为 `200 OK` 。

这也更好地解释了 `bulk` 请求是独立的，每一条的失败与否 都不会影响到其他的请求。

能省就省

或许你在批量导入大量的数据到相同的 `index` 以及 `type` 中。每次都去指定每个文档的metadata是完全没有必要的。在 `mget` API中， `bulk` 请求可以在URL中声明 `/_index` 或者 `/_index/_type` ：

```
POST /website/_bulk
{ "index": { "_type": "log" }}
{ "event": "User logged in" }
```

你依旧可以在metadata行中使用 `_index` 以及 `_type` 来重写数据，未声明的将会使用URL中的配置作为默认值：

```
POST /website/log/_bulk
{ "index": {}}
{ "event": "User logged in" }
{ "index": { "_type": "blog" }}
{ "title": "Overriding the default type" }
```

最大有多大？

整个数据将会被处理它的节点载入内存中，所以如果请求量很大的话，留给其他请求的内存空间将会很少。 `bulk` 应该有一个最佳的限度。超过这个限制后，性能不但不会提升反而可能会造成宕机。

最佳的容量并不是一个确定的数值，它取决于你的硬件，你的文档大小以及复杂性，你的索引以及搜索的负载。幸运的是，这个平衡点 很容易确定：

试着去批量索引越来越多的文档。当性能开始下降的时候，就说明你的数据量太大了。一般比较好初始数量级是1000到5000个文档，或者你的文档很大，你就可以试着减小队列。 有的时候看看批量请求的物理大小是很有帮助的。1000个1KB的文档和1000个1MB的文档的差距将会是天差地别的。比较好的初始批量容量是5-15MB。

总结

现在你应该知道如何作为分布式文档存储来使用Elasticsearch。你可以对文档进行存储，更新，获取，删除操作，而且你还知道该如何安全的执行这些操作。这已经非常有用处了，即使我们现在仍然没有尝试更激动人心的方面 -- 在文档中进行查询操作。不过我们先探讨下分布式环境中Elasticsearch安全管理你的文档所使用的内部过程。

分布式文档存储

本章将会在主要章节翻译结束后再继续翻译

In the last chapter, we looked at all the ways to put data into your index and then retrieve it. But we glossed over many technical details surrounding how the data is distributed and fetched from the cluster. This separation is done on purpose -- you don't really need to know how data is distributed to work with Elasticsearch. It just works.

In this chapter, we are going to dive into those internal, technical details to help you understand how your data is stored in a distributed system.

> 内容警告

The information presented below is for your interest. You are not required to understand and remember all the detail in order to use Elasticsearch. The options that are discussed are for advanced users only.

Read the section to gain a taste for how things work, and to know where the information is in case you need to refer to it in the future, but don't be overwhelmed by the detail.

将文档路由到从库中

当你索引一个文档，它被保存在单个的主分片上，Elasticsearch如何知道文档属于哪个分片呢？ 当我们创建一个新文档，它如何知道应该存储在分片1还是分片2上呢？

这个过程不能是随机的，因为我们将来需要取回该文档。 事实上，它是由一个非常简单的公式来决定的:

```
分片 = hash(routing) % 主分片数量
```

`routing` 值可以是任何的字符串， 默认是文档的 `_id` ，但也可以设置成一个自定义的值。 `routing` 字符串被传递到一个哈希函数以生成一个数字，然后除以索引的主分片的数量 得到余数 *remainder*. 余数将总是在 `0` 到 `主分片数量 - 1` 之间, 它告诉了我们用以存放 一个特定文档的分片编号。

这解释了为什么主分片的数量只能在索引创建时设置、而且不能修改。 如果主分片的数量一旦在日后进行了修改，所有之前的路由值都会无效，文档再也无法被找到。

所有文档 APIs (`get` , `index` , `delete` , `bulk` , `update` 和 `mget`) 都可以接受 `routing` 参数，用以自定义 文档-到-分片 的映射。 自定义的路由将用于确保所有的文档 -- 例如属于同一用户的所有文档 -- 保存在相同的分片上。 我们将在 `<<扩展>>` 中详细讨论你为什么希望这么做。

主从库之间是如何通信的

For explanation purposes, let's imagine that we have a cluster consisting of 3 nodes. It contains one index called `blogs` which has two primary shards. Each primary shard has two replicas. Copies of the same shard are never allocated to the same node, so our cluster looks something like <>.

[[img-distrib]] .A cluster with three nodes and one index image::images/04-01_index.png["A cluster with three nodes and one index"]

We can send our requests to any node in the cluster. Every node is fully capable of serving any request. Every node knows the location of every document in the cluster and so can forward requests directly to the required node. In the examples below, we will send all of our requests to `Node 1`, which we will refer to as the *requesting node*.

TIP: When sending requests, it is good practice to round-robin through all the nodes in the cluster, in order to spread the load.

创建、索引、删除文档

Create, index and delete requests are *write* operations, which must be successfully completed on the primary shard before they can be copied to any associated replica shards.

[[img-distrib-write]] .Creating, indexing or deleting a single document image::images/04-02_write.png["Creating, indexing or deleting a single document"]

Below we list the sequence of steps necessary to successfully create, index or delete a document on both the primary and any replica shards, as depicted in <>:

1. The client sends a create, index or delete request to `Node_1`.
2. The node uses the document's `_id` to determine that the document belongs to shard `0`. It forwards the request to `Node 3`, where the primary copy of shard `0` is currently allocated.
3. `Node 3` executes the request on the primary shard. If it is successful, it forwards the request in parallel to the replica shards on `Node 1` and `Node 2`. Once all of the replica shards report success, `Node 3` reports success to the requesting node, which reports success to the client.

By the time the client receives a successful response, the document change has been executed on the primary shard and on all replica shards. Your change is safe.

There are a number of optional request parameters which allow you to influence this process, possibly increasing performance at the cost of data security. These options are seldom used because Elasticsearch is already fast, but they are explained here for the sake of completeness.

`replication` ::

+

The default value for replication is `sync`. This causes the primary shard to wait for successful responses from the replica shards before returning.

If you set `replication` to `async`, then it will return success to the client as soon as the request has been executed on the primary shard. It will still forward the request to the replicas, but you will not know if the replicas succeeded or not.

It is advisable to use the default `sync` replication as it is possible to overload Elasticsearch by sending too many requests without waiting for their

completion.

`consistency` ::

+

By default, the primary shard requires a *quorum* or majority of shard copies (where a shard copy can be a primary or a replica shard) to be available before even attempting a write operation. This is to prevent writing data to the ``wrong side'' of a network partition. A quorum is defined as:

```
int( (primary + number_of_replicas) / 2 ) + 1
```

The allowed values for `consistency` are `one` (just the primary shard), `all` (the primary and all replicas) or the default `quorum` or majority of shard copies.

Note that the `number_of_replicas` is the number of replicas *specified* in the index settings, not the number of replicas that are currently active. If you have specified that an index should have 3 replicas then a quorum would be:

```
int( (primary + 3 replicas) / 2 ) + 1 = 3
```

But if you only start 2 nodes, then there will be insufficient active shard copies to satisfy the quorum and you will be unable to index or delete any documents.

--

`timeout` ::

What happens if insufficient shard copies are available? Elasticsearch waits, in the hope that more shards will appear. By default it will wait up to one minute. If you need to, you can use the `timeout` parameter to make it abort sooner: `100` is 100 milliseconds, `30s` is 30 seconds.

[NOTE]

A new index has `1` replica by default, which means that two active shard copies *should* be required in order to satisfy the need for a `quorum`. However, these default settings would prevent us from doing anything useful with a single-node cluster. To avoid this problem, the requirement for

a quorum is only enforced when `number_of_replicas` is greater than `1`.

获取一个文档

A document can be retrieved from a primary shard or from any of its replicas.

[[img-distrib-read]] .Retrieving a single document image::images/04-03_get.png["Retrieving a single document"]

Below we list the sequence of steps to retrieve a document from either a primary or replica shard, as depicted in <>:

1. The client sends a get request to `Node 1`.
2. The node uses the document's `_id` to determine that the document belongs to shard `0`. Copies of shard `0` exist on all three nodes. On this occasion, it forwards the request to `Node 2`.
3. `Node 2` returns the document to `Node 1` which returns the document to the client.

For read requests, the requesting node will choose a different shard copy on every request in order to balance the load -- it round-robins through all shard copies.

It is possible that a document has been indexed on the primary shard but has not yet been copied to the replica shards. In this case a replica might report that the document doesn't exist, while the primary would have returned the document successfully.

更新文档中的一部分

The `update` API combines the read and write patterns explained above.

[[img-distrib-update]] .Partial updates to a document image::images/04-04_update.png["Partial updates to a document"]

Below we list the sequence of steps used to perform a partial update on a document, as depicted in <>:

1. The client sends an update request to `Node_1` .
2. It forwards the request to `Node 3` , where the primary shard is allocated.
3. `Node 3` retrieves the document from the primary shard, changes the JSON in the `_source` field, and tries to reindex the document on the primary shard. If the document has already been changed by another process, it retries step 3 up to `retry_on_conflict` times, before giving up.
4. If `Node 3` has managed to update the document successfully, it forwards the new version of the document in parallel to the replica shards on `Node 1` and `Node 2` to be reindexed. Once all replica shards report success, `Node 3` reports success to the requesting node, which reports success to the client.

The `update` API also accepts the `routing` , `replication` , `consistency` and `timeout` parameters that are explained in <>.

> ### 基于文档的复制

When a primary shard forwards changes to its replica shards, it doesn't forward the update request. Instead it forwards the new version of the full document. Remember that these changes are forwarded to the replica shards asynchronously and there is no guarantee that they will arrive in the same order that they were sent. If Elasticsearch forwarded just the change, it is possible that changes would be applied in the wrong order, resulting in a corrupt document.

多文档模式

The patterns for the `mget` and `bulk` APIs are similar to those for individual documents. The difference is that the requesting node knows in which shard each document lives. It breaks up the multi-document request into a multi-document request *per shard*, and forwards these in parallel to each participating node.

Once it receives answers from each node, it collates their responses into a single response, which it returns to the client.

[[img-distrib-mget]] .Retrieving multiple documents with `mget` image::images/04-05_mget.png["Retrieving multiple documents with mget"]

Below we list the sequence of steps necessary to retrieve multiple documents with a single `mget` request, as depicted in <>:

1. The client sends an `mget` request to `Node_1`.
2. `Node_1` builds a multi-get request per shard, and forwards these requests in parallel to the nodes hosting each required primary or replica shard. Once all replies have been received, `Node_1` builds the response and returns it to the client.

A `routing` parameter can be set for each document in the `docs` array, and the `preference` parameter can be set for the top-level `mget` request.

[[img-distrib-bulk]] .Multiple document changes with `bulk` image::images/04-06_bulk.png["Multiple document changes with bulk"]

Below we list the sequence of steps necessary to execute multiple `create`, `index`, `delete` and `update` requests within a single `bulk` request, as depicted in <>:

1. The client sends a `bulk` request to `Node_1`.
2. `Node_1` builds a bulk request per shard, and forwards these requests in parallel to the nodes hosting each involved primary shard.
3. The primary shard executes each action serially, one after another. As each action succeeds, the primary forwards the new document (or deletion) to its replica shards in parallel, then moves on to the next action. Once all replica shards report success for all actions, the node reports success to the requesting node, which collates the responses and returns them to the client.

The `bulk` API also accepts the `replication` and `consistency` parameters at the top-level for the whole `bulk` request, and the `routing` parameter in the metadata for each request.

Why the funny format?

When we learned about Bulk requests earlier in <>, you may have asked yourself: `` `Why does the `` bulk `API require the funny format with the newline characters, instead of just sending the requests wrapped in a JSON array, like the` mget` API?"

To answer this, we need to explain a little background:

Each document referenced in a bulk request may belong to a different primary shard, each of which may be allocated to any of the nodes in the cluster. This means that every *action* inside a `bulk` request needs to be forwarded to the correct shard on the correct node.

If the individual requests were wrapped up in a JSON array, that would mean that we would need to:

- parse the JSON into an array (including the document data, which can be very large)
- look at each request to determine which shard it should go to
- create an array of requests for each shard
- serialize these arrays into the internal transport format
- send the requests to each shard

It would work, but would need a lot of RAM to hold copies of essentially the same data, and would create many more data structures that the JVM would have to spend time garbage collecting.

Instead, Elasticsearch reaches up into the networking buffer, where the raw request has been received and reads the data directly. It uses the newline characters to identify and parse just the small *action/metadata* lines in order to decide which shard should handle each request.

These raw requests are forwarded directly to the correct shard. There is no redundant copying of data, no wasted data structures. The entire request process is handled in the smallest amount of memory possible.

搜索 — 基本工具

到目前为止，我们已经学习了Elasticsearch的分布式NOSQL文档存储，我们可以直接把JSON文档扔到Elasticsearch中，然后直接通过ID来进行调取。但是Elasticsearch真正的强大之处在于将混乱变得有意义——将大数据变成大量的信息。

这也是我们使用JSON文档而不是无规则数据的原因。Elasticsearch不仅仅只是存储文档，同时它还索引了这些文档以便搜索。文档中每一个字段都被索引并且可以被查询。不仅如此，在一个查询中，Elasticsearch可以使用所有索引，并且以惊人的速度返回结果。这是传统数据库永远也不能企及的。

这个搜索可以是：

- 类似于 `年龄` 、 `性别` 、 `加入日期` 等结构化数据，类似于在SQL中进行查询。
- 全文搜索，查找整个文档中匹配关键字的内容，并根据相关性
- 或者结合两者。

虽然很多搜索操作是安装好Elasticsearch就可以用的，但是想发挥它的潜力，你需要明白以下内容：

名字	说明
映射 *(Mapping)*	每个字段中的数据如何被解释
统计 *(Analysis)*	可搜索的全文是如何被处理的
查询 *(Query DSL)*	Elasticsearch使用的灵活强的查询语言

上述的每一个内容都是一个大的主题，我们将会在之后的《深入搜索》中详细探讨它们。 本章中我们将针对先去介绍它们三个的基本概念 —— 已经足够能帮助你理解搜索是如何运作的了。

我们将向你介绍 `search` API的简单实用方式。

> ## 测试数据

我们本章使用的文档可以在下面的git中找到：https://gist.github.com/clintongormley/8579281

你可以下载然后导入到你的shell中以方便你的学习使用。

空白搜索

搜索API最常用的一种形式就是空白搜索，也就是不加任何查询条件的，只是返回集群中所有文档的搜索。

```
GET /_search
```

返回内容如下（有删减）：

```
{
   "hits" : {
      "total" :       14,
      "hits" : [
        {
          "_index":   "us",
          "_type":    "tweet",
          "_id":      "7",
          "_score":   1,
          "_source": {
             "date":    "2014-09-17",
             "name":    "John Smith",
             "tweet":   "The Query DSL is really powerful and flexible",
             "user_id": 2
          }
       },
        ... 9 个结果被隐藏 ...
      ],
      "max_score" :   1
   },
   "took" :           4,
   "_shards" : {
      "failed" :      0,
      "successful" :  10,
      "total" :       10
   },
   "timed_out" :      false
}
```

`hits`

返回内容中最重要的内容就是 `hits` ，它指明了匹配查询的文档的 `总数` ， `hits` 数组里则会包含前十个匹配文档——也就是搜索结果。

`hits` 数组中的每一条结果都包含了文档的 `_index` , `_type` 以及 `_id` 信息，以及 `_source` 字段。这也就意味着你可以直接从搜索结果中获取到整个文档的内容。这与其他搜索引擎只返回给你文档编号，还需要自己去获取文档是截然不同的。

每一个元素还拥有一个 `_score` 字段。这个是相关性评分，这个数值表示当前文档与查询的匹配程度。通常来说，搜索结果会先返回最匹配的文档，也就是说它们会按照 `_score` 由高至低进行排列。在这个例子中，我们并没有声明任何查询，因此 `_score` 就都会返回 `1`

`max_score` 数值会显示所有匹配文档中的 `_score` 的最大值。

`took`

`took` 数值告诉我们执行这次搜索请求所耗费的时间有多少毫秒。

`shards`

`_shards` 告诉了我们参与查询分片的总数，以及有多少 `successful` 和 `failed` 。通常情况下我们是不会得到失败的反馈，但是有的时候它会发生。如果我们的服务器突然出现了重大事故，然后我们丢失了同一个分片中主从两个版本的数据。在查询请求中，无法提供可用的备份。这种情况下，Elasticsearch就会返回`failed提示，但是它还会继续返回剩下的内容。

`timeout`

`timed_out` 数值告诉了我们查询是否超时。通常，搜索请求不会超时。如果相比完整的结果你更需要的是快速的响应时间，这是你可以指定 `timeout` 值，例如 `10` 、 `"10ms"` （10毫秒）或者 `"1s"` （1秒钟）：

```
GET /_search?timeout=10ms
```

Elasticsearch会尽可能地返回你指定时间内它所查到的内容。

> ### Timeout并不是终止者
>
> 这里应该强调一下 `timeout` 并不会终止查询，它只是会在你指定的时间内返回当时已经查询到的数据，然后关闭连接。在后台，其他的查询可能会依旧继续，尽管查询结果已经被返回了。
>
> 使用超时是因为你要保障你的品质，并不是因为你需要终止你的查询。

多索引，多类型

你是否注意到了《空白搜索》一章节的文档中包含了很多不同的类型 —— `user` 与 `tweet` ，它们也分别来自 `us` 、 `gb` 这两个不同的索引？

当我们没有特别指定一个索引或者类型的时候，我们将会搜索整个集群中的所有文档。Elasticsearch会把搜索请求转发给集群中的每一个主从分片，然后按照结果的相关性得到前十名，并将它们返回给我们。

然而，往往我们只需要在某一个特定的索引的几个类型中进行搜索。我们可以通过在URL中定义它来实现这个功能：

URL	说明
`/_search`	搜索所有的索引和类型
`/gb/_search`	搜索索引 `gb` 中的所有类型
`/gb,us/_search`	搜索索引 `gb` 以及 `us` 中的所有类型
`/g*,u*/_search`	搜索所有以 `g` 或 `u` 开头的索引中的所有类型
`/gb/user/_search`	搜索索引 `gb` 中类型 `user` 内的所有文档
`/gb,us/user,tweet/_search`	搜索索引 `gb` 和 索引 `us` 中类型 `user` 以及类型 `tweet` 内的所有文档
`/_all/user,tweet/_search`	搜索所有索引中类型为 `user` 以及 `tweet` 内的所有文档

当你在一个索引中搜索的时候，Elasticsearch或将你的搜索请求转发给相应索引中的所有主从分片，然后收集每一个分片的结果。在多个索引中搜索也是相同的流程，只不过是增加了一些参与分片。

> ## 重要提示
>
> 搜索一个拥有五个主分片的索引与搜索五个都只拥有一个主分片是完全一样的。

在后面，你将会了解到如何利用这一点，来根据你的需要灵活打造系统。

分页

在《空白搜索》一节中，搜索结果告诉我们在集群中共有14个文档匹配我们的（空白）查询。但是在 `hits` 数组中只有10个文档。我们怎样才能看到其他的呢？

与SQL使用 `LIMIT` 来控制单“页”数量类似，Elasticsearch使用的是 `from` 以及 `size` 两个参数：

参数	说明
`size`	每次返回多少个结果，默认值为 `10`
`from`	忽略最初的几条结果，默认值为 `0`

假设每页显示5条结果，那么1至3页的请求就是：

```
GET /_search?size=5
GET /_search?size=5&from=5
GET /_search?size=5&from=10
```

当心不要一次请求过多或者页码过大的结果。它们会在返回前排序。一个请求会经过多个分片。每个分片都会生成自己的排序结果。然后再进行集中整理，以确保最终结果的正确性。

> ## 分布式系统中的大页码页面

为了说明白为什么页码过大的请求会产生问题，我们就先预想一下我们在搜索一个拥有5个主分片的索引。当我们请求第一页搜索的时候，每个分片产生自己前十名，然后将它们返回给请求节点，然后这个节点会将50条结果重新排序以产生最终的前十名。

现在想想一下我们想获得第1,000页，也就是第10,001到第10,010条结果，与之前同理，每一个分片都会先产生自己的前10,010名，然后请求节点统一处理这50,050条结果，然后再丢弃掉其中的50,040条！

现在你应该明白了，在分布式系统中，大页码请求所消耗的系统资源是呈指数式增长的。这也是为什么网络搜索引擎不会提供超过1,000条搜索结果的原因。

> ## TIP

在《重索引》一章中，我们将详细探讨如何才能高效地获取大量数据。

精简 搜索

搜索的API分为两种：其一是通过参数来传递查询的“精简版”查询语句（*query string*），还有一种是通过JSON来传达丰富的查询的完整版请求体（*request body*），这种搜索语言被称为查询DSL。

查询语句在行命令中运行点对点查询的时候非常实用。比如我想要查询所有 `tweet` 类型中，所有 `tweet` 字段为 `"elasticsearch"` 的文档：

```
GET /_all/tweet/_search?q=tweet:elasticsearch
```

下一个查询是想要寻找 `name` 字段为 `"john"` 且 `tweet` 字段为 `"mary"` 的文档，实际的查询就是：

```
+name:john +tweet:mary
```

但是经过百分号编码（*percent encoding*）处理后，会让它看起来稍显神秘:

```
GET /_search?q=%2Bname%3Ajohn+%2Btweet%3Amary
```

前缀 `"+"` 表示必须要满足我们的查询匹配条件，而前缀 `"-"` 则表示绝对不能匹配条件。没有 `+` 或者 `-` 的表示可选条件。匹配的越多，文档的相关性就越大。

字段 `_all`

下面这条简单的搜索将会返回所有包含 `"mary"` 字符的文档：

```
GET /_search?q=mary
```

在之前的例子中，我们搜索 `tweet` 或者 `name` 中的文字。然而，搜索的结果显示 `"mary"` 在三个不同的字段中：

- 用户的名字为"Mary"
- 6个"Mary"发送的推文
- 1个"@mary"

那么Elasticsearch是如何找到三个不同字段中的内容呢？

当我们在索引一个文档的时候，Elasticsearch会将所有字段的数值都汇总到一个大的字符串中，并将它索引成一个特殊的字段 `_all` ：

```
{
    "tweet":    "However did I manage before Elasticsearch?",
    "date":     "2014-09-14",
    "name":     "Mary Jones",
    "user_id":  1
}
```

就好像我们已经添加了一个叫做 `_all` 的字段：

```
"However did I manage before Elasticsearch? 2014-09-14 Mary Jones 1"
```

除非指定了字段名，不然查询语句就会搜索字段 `_all` 。

TIP: 在你刚开始创建程序的时候你可能会经常使用 `_all` 这个字段。但是慢慢的，你可能就会在请求中指定字段。当字段 `_all` 已经没有使用价值的时候，那就可以将它关掉。之后的《字段all》一节中将会有介绍

更加复杂的查询

再实现一个查询：

- 字段 `name` 包含 `"mary"` 或 `"john"`
- `date` 大于 `2014-09-10`
- `_all` 字段中包含 `"aggregations"` 或 `"geo"`

```
+name:(mary john) +date:>2014-09-10 +(aggregations geo)
```

最终处理完的语句可读性可能很差：

```
?q=%2Bname%3A(mary+john)+%2Bdate%3A%3E2014-09-10+%2B(aggregations+geo)
```

正如你所看到的，这个简明查询语句是出奇的强大。在查询语句语法中，有关于它详细的介绍。借助它我们就可以在开发的时候提高很多效率。

不过，你也会发现简洁带来的易读性差和难以调试，以及它的脆弱：当其中出现 `-` , `:` , `/` 或者 `"` 时，它就会返回错误提示。

最后要提一句，任何用户都可以通过查询语句来访问臃肿的查询，或许会得到一些私人的信息，或许会通过大量的运算将你的集群压垮！

TIP

出于以上原因，我们不建议你将查询语句直接暴露给用户，除非是你信任的可以访问数据与集群的权限用户。

与此同时，在生产环境中，我们经常会使用到查询语句。在了解更多关于搜索的知识前，我们先来看一下它是怎样运作的。

映射与统计

当我们在进行搜索的事情，我们会发现有一些奇怪的事情。比如有一些内容似乎是被打破了：在我们的索引中有12条推文，中有一个包含了 `2014-09-15` 这个日期，但是看看下面的查询结果中的总数量：

```
GET /_search?q=2014              # 12 results
GET /_search?q=2014-09-15        # 12 results !
GET /_search?q=date:2014-09-15   # 1  result
GET /_search?q=date:2014         # 0  results !
```

为什么我们使用字段 `_all` 搜索全年就会返回所有推文，而使用字段 `date` 搜索年份却没有结果呢？为什么使用两者所得到的结果是不同的？

推测大概是因为我们的数据在 `_all` 和 `date` 在索引时没有被相同处理。我们来看看Elasticsearch是如何处理我们的文档结构的。我们可以对 `gb` 的 `tweet` 使用*mapping*请求：

```
GET /gb/_mapping/tweet
```

我们得到：

```
{
   "gb": {
      "mappings": {
         "tweet": {
            "properties": {
               "date": {
                  "type": "date",
                  "format": "dateOptionalTime"
               },
               "name": {
                  "type": "string"
               },
               "tweet": {
                  "type": "string"
               },
               "user_id": {
                  "type": "long"
               }
            }
         }
      }
   }
}
```

Elasticsearch会根据系统自动判断字段类型并生成一个映射。返回结果告诉我们 `date` 字段被识别成了 `date` 类型。 `_all` 没有出现是因为他是默认字段，但是我们知道字段 `_all` 实际上是 `string` 类型的。

所以类型为 `date` 的字段和类型为 `string` 的字段的索引方式是不同的。

So fields of type `date` and fields of type `string` are indexed differently, and can thus be searched differently. That's not entirely surprising. You might expect that each of the core data types -- strings, numbers, booleans and dates -- might be indexed slightly differently. And this is true: there are slight differences.

But by far the biggest difference is actually between fields that represent *exact values* (which can include `string` fields) and fields that represent *full text*. This distinction is really important -- it's the thing that separates a search engine from all other databases.

精确值与全文

Data in Elasticsearch can be broadly divided into two types: *exact values* and *full text*.

Exact values are exactly what they sound like. Examples would be a date or a user ID, but can also include exact strings like a username or an email address. The exact value `"Foo"` is not the same as the exact value `"foo"`. The exact value `2014` is not the same as the exact value `2014-09-15`.

Full text, on the other hand, refers to textual data -- usually written in some human language -- like the text of a tweet or the body of an email.

Full text is often referred to as ``unstructured data'', which is a misnomer -- natural language is highly structured. The problem is that the rules of natural languages are complex which makes them difficult for computers to parse correctly. For instance, consider this sentence:

```
May is fun but June bores me.
```

Does it refer to months or to people?

Exact values are easy to query. The decision is binary -- a value either matches the query, or it doesn't. This kind of query is easy to express with SQL:

```
WHERE name    = "John Smith"
  AND user_id = 2
  AND date    > "2014-09-15"
```

Querying full text data is much more subtle. We are not just asking `Does this document match the query'', but` How *well* does this document match the query?" In other words, how *relevant* is this document to the given query?

We seldom want to match the whole full text field exactly. Instead, we want to search *within* text fields. Not only that, but we expect search to understand our *intent*:

- a search for `"UK"` should also return documents mentioning the `"United Kingdom"`
- a search for `"jump"` should also match `"jumped"`, `"jumps"`, `"jumping"` and perhaps even `"leap"`

- `"johnny walker"` should match `"Johnnie Walker"` and `"johnnie depp"` should match `"Johnny Depp"`
- `"fox news hunting"` should return stories about hunting on Fox News, while `"fox hunting news"` should return news stories about fox hunting.

In order to facilitate these types of queries on full text fields, Elasticsearch first *analyzes* the text, then uses the results to build an *inverted index*. We will discuss the inverted index and the analysis process in the next two sections.

反向索引

Elasticsearch uses a structure called an *inverted index* which is designed to allow very fast full text searches. An inverted index consists of a list of all the unique words that appear in any document, and for each word, a list of the documents in which it appears.

For example, let's say we have two documents, each with a `content` field containing:

1. ``The quick brown fox jumped over the lazy dog''
2. ``Quick brown foxes leap over lazy dogs in summer''

To create an inverted index, we first split the `content` field of each document into separate words (which we call *terms* or *tokens*), create a sorted list of all the unique terms, then list in which document each term appears. The result looks something like this:

```
Term      Doc_1  Doc_2
-------------------------
Quick   |       |  X
The     |   X   |
brown   |   X   |  X
dog     |   X   |
dogs    |       |  X
fox     |   X   |
foxes   |       |  X
in      |       |  X
jumped  |   X   |
lazy    |   X   |  X
leap    |       |  X
over    |   X   |  X
quick   |   X   |
summer  |       |  X
the     |   X   |
------------------------
```

Now, if we want to search for `"quick brown"` we just need to find the documents in which each term appears:

```
Term      Doc_1  Doc_2
-------------------------
brown   |   X   |  X
quick   |   X   |
------------------------
Total   |   2   |  1
```

Both documents match, but the first document has more matches than the second. If we apply a naive *similarity algorithm* which just counts the number of matching terms, then we can say that the first document is a better match -- is *more relevant* to our query -- than the second document.

But there are a few problems with our current inverted index:

1. `"Quick"` and `"quick"` appear as separate terms, while the user probably thinks of them as the same word.
2. `"fox"` and `"foxes"` are pretty similar, as are `"dog"` and `"dogs"` -- they share the same root word.
3. `"jumped"` and `"leap"`, while not from the same root word, are similar in meaning -- they are synonyms.

With the above index, a search for `"+Quick +fox"` wouldn't match any documents. (Remember, a preceding `+` means that the word must be present). Both the term `"Quick"` and the term `"fox"` have to be in the same document in order to satisfy the query, but the first doc contains `"quick fox"` and the second doc contains `"Quick foxes"`.

Our user could reasonably expect both documents to match the query. We can do better.

If we normalize the terms into a standard format, then we can find documents that contain terms that are not exactly the same as the user requested, but are similar enough to still be relevant. For instance:

1. `"Quick"` can be lowercased to become `"quick"`.
2. `"foxes"` can be *stemmed* -- reduced to its root form -- to become `"fox"`. Similarly `"dogs"` could be stemmed to `"dog"`.
3. `"jumped"` and `"leap"` are synonyms and can be indexed as just the single term `"jump"`.

Now the index looks like this:

```
Term      Doc_1  Doc_2
-------------------------
brown   |   X   |  X
dog     |   X   |  X
fox     |   X   |  X
in      |       |  X
jump    |   X   |  X
lazy    |   X   |  X
over    |   X   |  X
quick   |   X   |  X
summer  |       |  X
the     |   X   |  X
------------------------
```

But we're not there yet. Our search for `"+Quick +fox"` would *still* fail, because we no longer have the exact term `"Quick"` in our index. However, if we apply the same normalization rules that we used on the `content` field to our query string, it would become a query for `"+quick +fox"`, which would match both documents!

IMPORTANT: This is very important. You can only find terms that actually exist in your index, so: *both the indexed text and and query string must be normalized into the same form*.

This process of tokenization and normalization is called *analysis*, which we discuss in the next section.

[[analysis-intro]] === Analysis and analyzers

Analysis is the process of:

- first, tokenizing a block of text into individual *terms* suitable for use in an inverted index,
- then normalizing these terms into a standard form to improve their ``searchability'' or *recall*.

This job is performed by *analyzers*. An *analyzer* is really just a wrapper which combines three functions into a single package:

Character filters::

```
First, the string is passed through any _character filters_ in turn. Their
job is to tidy up the string before tokenization. A character filter could
be used to strip out HTML, or to convert `"&"` characters to the word
`"and"`.
```

Tokenizer::

Next, the string is tokenized into individual terms by a *tokenizer*. A simple tokenizer might split the text up into terms whenever it encounters whitespace or punctuation.

Token filters::

Last, each term is passed through any *token filters* in turn, which can change terms (eg lowercasing `"Quick"`), remove terms (eg stopwords like `"a"` , `"and"` , `"the"` etc) or add terms (eg synonyms like `"jump"` and `"leap"`)

Elasticsearch provides many character filters, tokenizers and token filters out of the box. These can be combined to create custom analyzers suitable for different purposes. We will discuss these in detail in <>.

==== Built-in analyzers

However, Elasticsearch also ships with a number of pre-packaged analyzers that you can use directly. We list the most important ones below and, to demonstrate the difference in behaviour, we show what terms each would produce from this string:

```
"Set the shape to semi-transparent by calling set_trans(5)"
```

Standard analyzer::

The standard analyzer is the default analyzer that Elasticsearch uses. It is the best general choice for analyzing text which may be in any language. It splits the text on *word boundaries*, as defined by the http://www.unicode.org/reports/tr29/[Unicode Consortium],

and removes most punctuation. Finally, it lowercases all terms. It would produce: + set, the, shape, to, semi, transparent, by, calling, set_trans, 5

Simple analyzer::

The simple analyzer splits the text on anything that isn't a letter, and lowercases the terms. It would produce: + set, the, shape, to, semi, transparent, by, calling, set, trans

Whitespace analyzer::

The whitespace analyzer splits the text on whitespace. It doesn't lowercase. It would produce: + Set, the, shape, to, semi-transparent, by, calling, set_trans(5)

Language analyzers::

Language-specific analyzers are available for many languages. They are able to take the peculiarities of the specified language into account. For instance, the `english` analyzer comes with a set of English stopwords -- common words like `and` or `the` which don't have much impact on relevance -- which it removes, and it is able to *stem* English words because it understands the rules of English grammar. + The `english` analyzer would produce the following: + set, shape, semi, transpar, call, set_tran, 5 + Note how `"transparent"`, `"calling"`, and `"set_trans"` have been stemmed to their root form.

==== When analyzers are used

When we *index* a document, its full text fields are analyzed into terms which are used to create the inverted index. However, when we *search* on a full text field, we need to pass the query string through the *same analysis process*, to ensure that we are searching for terms in the same form as those that exist in the index.

Full text queries, which we will discuss later, understand how each field is defined, and so they can do the right thing:

- When you query a *full text* field, the query will apply the same analyzer to the query string to produce the correct list of terms to search for.

- When you query an *exact value* field, the query will not analyze the query string, but instead search for the exact value that you have specified.

Now you can understand why the queries that we demonstrated at the <> return what they do:

- The `date` field contains an exact value: the single term `"2014-09-15"`.
- The `_all` field is a full text field, so the analysis process has converted the date into the three terms: `"2014"`, `"09"` and `"15"`.

When we query the `_all` field for `2014`, it matches all 12 tweets, because all of them contain the term `2014`:

[source,sh]

GET /_search?q=2014 # 12 results

// SENSE: 052_Mapping_Analysis/25_Data_type_differences.json

When we query the `_all` field for `2014-09-15`, it first analyzes the query string to produce a query which matches *any* of the terms `2014`, `09` or `15`. This also matches all 12 tweets, because all of them contain the term `2014`:

[source,sh]

GET /_search?q=2014-09-15 # 12 results !

// SENSE: 052_Mapping_Analysis/25_Data_type_differences.json

When we query the `date` field for `2014-09-15`, it looks for that *exact* date, and finds one tweet only:

[source,sh]

GET /_search?q=date:2014-09-15 # 1 result

// SENSE: 052_Mapping_Analysis/25_Data_type_differences.json

When we query the `date` field for `2014`, it finds no documents because none contain that exact date:

[source,sh]

GET /_search?q=date:2014 # 0 results !

// SENSE: 052_Mapping_Analysis/25_Data_type_differences.json

[[analyze-api]] ==== Testing analyzers

Especially when you are new to Elasticsearch, it is sometimes difficult to understand what is actually being tokenized and stored into your index. To better understand what is going on, you can use the `analyze` API to see how text is analyzed. Specify which analyzer to use in the query string parameters, and the text to analyze in the body:

[source,js]

GET /_analyze?analyzer=standard

Text to analyze

// SENSE: 052_Mapping_Analysis/40_Analyze.json

Each element in the result represents a single term:

[source,js]

{ "tokens": [{ "token": "text", "start_offset": 0, "end_offset": 4, "type": "", "position": 1 }, { "token": "to", "start_offset": 5, "end_offset": 7, "type": "", "position": 2 }, { "token": "analyze", "start_offset": 8, "end_offset": 15, "type": "", "position": 3 }]

}

The `token` is the actual term that will be stored in the index. The `position` indicates the order in which the terms appeared in the original text. The `start_offset` and `end_offset` indicate the character positions that the original word occupied in the original string.

The `analyze` API is really useful tool for understanding what is happening inside Elasticsearch indices, and we will talk more about it as we progress.

==== Specifying analyzers

When Elasticsearch detects a new string field in your documents, it automatically configures it as a full text `string` field and analyzes it with the `standard` analyzer.

You don't always want this. Perhaps you want to apply a different analyzer which suits the language your data is in. And sometimes you want a string field to be just a string field -- to index the exact value that you pass in, without any analysis, such as a string user ID or an

internal status field or tag.

In order to achieve this, we have to configure these fields manually by specifying the *mapping*.

映射

As explained in <>, each document in an index has a *type*. Every type has its own *mapping* or *schema definition*. A mapping defines the fields within a type, the datatype for each field, and how the field should be handled by Elasticsearch. A mapping is also used to configure metadata associated with the type.

We discuss mappings in detail in <>. In this section we're going to look at just enough to get you started.

[[core-fields]] ==== Core simple field types

Elasticsearch supports the following simple field types:

[horizontal] String: :: `string` Whole number: :: `byte` , `short` , `integer` , `long` Floating point: :: `float` , `double` Boolean: :: `boolean` Date: :: `date`

When you index a document which contains a new field -- one previously not seen -- Elasticsearch will use <> to try to guess the field type from the basic datatypes available in JSON, using the following rules:

[horizontal] *JSON type:* :: *Field type:*

Boolean: `true` or `false` :: `"boolean"`

Whole number: `123` :: `"long"`

Floating point: `123.45` :: `"double"`

String, valid date: `"2014-09-15"` :: `"date"`

String: `"foo bar"` :: `"string"`

NOTE: This means that, if you index a number in quotes -- `"123"` it will be mapped as type `"string"` , not type `"long"` . However, if the field is already mapped as type `"long"` , then Elasticsearch will try to convert the string into a long, and throw an exception if it can't.

==== Viewing the mapping

We can view the mapping that Elasticsearch has for one or more types in one or more indices using the `/_mapping` endpoint. At the <> we already retrieved the mapping for type `tweet` in index `gb` :

[source,js]

GET /gb/_mapping/tweet

This shows us the mapping for the fields (called *properties*) that Elasticsearch generated dynamically from the documents that we indexed:

[source,js]

{ "gb": { "mappings": { "tweet": { "properties": { "date": { "type": "date", "format": "dateOptionalTime" }, "name": { "type": "string" }, "tweet": { "type": "string" }, "user_id": { "type": "long" } } } } }

}

[TIP]

Incorrect mappings, such as having an `age` field mapped as type `string` instead of `integer`, can produce confusing results to your queries.

Instead of assuming that your mapping is correct, check it!

[[custom-field-mappings]] ==== Customizing field mappings

The most important attribute of a field is the `type`. For fields other than `string` fields, you will seldom need to map anything other than `type`:

[source,js]

{ "number_of_clicks": { "type": "integer" }

}

Fields of type `"string"` are, by default, considered to contain full text. That is, their value will be passed through an analyzer before being indexed and a full text query on the field will pass the query string through an analyzer before searching.

The two most important mapping attributes for `string` fields are `index` and `analyzer`.

===== `index`

The `index` attribute controls how the string will be indexed. It can contain one of three values:

[horizontal] `analyzed` :: First analyze the string, then index it. In other words, index this field as full text.

`not_analyzed` :: Index this field, so it is searchable, but index the value exactly as specified. Do not analyze it.

`no` :: Don't index this field at all. This field will not be searchable.

The default value of `index` for a `string` field is `analyzed`. If we want to map the field as an exact value, then we need to set it to `not_analyzed`:

[source,js]

{ "tag": { "type": "string", "index": "not_analyzed" }

}

The other simple types -- `long`, `double`, `date` etc -- also accept the `index` parameter, but the only relevant values are `no` and `not_analyzed`, as their values are never analyzed.

===== `analyzer`

For `analyzed` string fields, use the `analyzer` attribute to specify which analyzer to apply both at search time and at index time. By default, Elasticsearch uses the `standard` analyzer, but you can change this by specifying one of the built-in analyzers, such as `whitespace`, `simple`, or `english`:

[source,js]

{ "tweet": { "type": "string", "analyzer": "english" }

}

In <> we will show you how to define and use custom analyzers as well.

==== Updating a mapping

You can specify the mapping for a type when you first create an index. Alternatively, you can add the mapping for a new type (or update the mapping for an existing type) later, using the `/_mapping` endpoint.

[IMPORTANT]

While you can *add* to an existing mapping, you can't *change* it. If a field already exists in the mapping, then it probably means that data from that field has already been indexed. If you were to change the field mapping, then

the already indexed data would be wrong and would not be properly searchable.

We can update a mapping to add a new field, but we can't change an existing field from `analyzed` to `not_analyzed`.

To demonstrate both ways of specifying mappings, let's first delete the `gb` index:

[source,sh]

DELETE /gb

// SENSE: 052_Mapping_Analysis/45_Mapping.json

Then create a new index, specifying that the `tweet` field should use the `english` analyzer:

[source,js]

PUT /gb <1> { "mappings": { "tweet" : { "properties" : { "tweet" : { "type" : "string", "analyzer": "english" }, "date" : { "type" : "date" }, "name" : { "type" : "string" }, "user_id" : { "type" : "long" } } } }

}

// SENSE: 052_Mapping_Analysis/45_Mapping.json

<1> This creates the index with the `mappings` specified in the body.

Later on, we decide to add a new `not_analyzed` text field called `tag` to the `tweet` mapping, using the `_mapping` endpoint:

[source,js]

PUT /gb/_mapping/tweet { "properties" : { "tag" : { "type" : "string", "index": "not_analyzed" } }

}

// SENSE: 052_Mapping_Analysis/45_Mapping.json

Note that we didn't need to list all of the existing fields again, as we can't change them anyway. Our new field has been merged into the existing mapping.

==== Testing the mapping

You can use the `analyze` API to test the mapping for string fields by name. Compare the output of these two requests:

[source,js]

GET /gb/_analyze?field=tweet Black-cats <1>

GET /gb/_analyze?field=tag

Black-cats <1>

// SENSE: 052_Mapping_Analysis/45_Mapping.json

<1> The text we want to analyze is passed in the body.

The `tweet` field produces the two terms `"black"` and `"cat"`, while the `tag` field produces the single term `"Black-cats"`. In other words, our mapping is working correctly.

[[complex-core-fields]] === Complex core field types

Besides the simple scalar datatypes that we mentioned above, JSON also has `null` values, arrays and objects, all of which are supported by Elasticsearch:

==== Multi-value fields

It is quite possible that we want our `tag` field to contain more than one tag. Instead of a single string, we could index an array of tags:

```
{ "tag": [ "search", "nosql" ]}
```

There is no special mapping required for arrays. Any field can contain zero, one or more values, in the same way as a full text field is analyzed to produce multiple terms.

By implication, this means that *all of the values of an array must be of the same datatype*. You can't mix dates with strings. If you create a new field by indexing an array, Elasticsearch will use the datatype of the first value in the array to determine the `type` of the new field.

The elements inside an array are not ordered. You cannot refer to `the first element'' or` the last element". Rather think of an array as a *bag of values*.

==== Empty fields

Arrays can, of course, be empty. This is the equivalent of having zero values. In fact, there is no way of storing a `null` value in Lucene, so a field with a `null` value is also considered to be an empty field.

These four fields would all be considered to be empty, and would not be indexed:

```
"empty_string":             "",
"null_value":               null,
"empty_array":              [],
"array_with_null_value":    [ null ]
```

==== Multi-level objects

The last native JSON datatype that we need to discuss is the *object* -- known in other languages as hashes, hashmaps, dictionaries or associative arrays.

Inner objects are often used to embed one entity or object inside another. For instance, instead of having fields called `user_name` and `user_id` inside our `tweet` document, we could write it as:

```js { "tweet": "Elasticsearch is very flexible", "user": { "id": "@johnsmith", "gender": "male", "age": 26, "name": { "full": "John Smith", "first": "John", "last": "Smith" } }

# }

==== Mapping for inner objects

Elasticsearch will detect new object fields dynamically and map them as type `object`, with each inner field listed under `properties`:

# [source,js]

{ "gb": { "tweet": { <1> "properties": { "tweet": { "type": "string" }, "user": { <2> "type": "object", "properties": { "id": { "type": "string" }, "gender": { "type": "string" }, "age": { "type": "long" }, "name": { <2> "type": "object", "properties": { "full": { "type": "string" }, "first": { "type": "string" }, "last": { "type": "string" } } } } } } } }

# }

<1> Root object.

<2> Inner objects.

The mapping for the `user` and `name` fields have a similar structure to the mapping for the `tweet` type itself. In fact, the `type` mapping is just a special type of `object` mapping, which we refer to as the *root object*. It is just the same as any other object, except that it has some special top-level fields for document metadata, like `_source`, the `_all` field etc.

==== How inner objects are indexed

Lucene doesn't understand inner objects. A Lucene document consists of a flat list of key-value pairs. In order for Elasticsearch to index inner objects usefully, it converts our document into something like this:

# [source,js]
```

{ "tweet": [elasticsearch, flexible, very], "user.id": [@johnsmith], "user.gender": [male], "user.age": [26], "user.name.full": [john, smith], "user.name.first": [john], "user.name.last": [smith]

}

Inner fields can be referred to by name, eg `"first"`. To distinguish between two fields that have the same name we can use the full *path*, eg `"user.name.first"` or even the `type` name plus the path: `"tweet.user.name.first"`.

NOTE: In the simple flattened document above, there is no field called `user` and no field called `user.name`. Lucene only indexes scalar or simple values, not complex datastructures.

==== Arrays of inner objects

Finally, consider how an array containing inner objects would be indexed. Let's say we have a `followers` array which looks like this:

[source,js]

{ "followers": [{ "age": 35, "name": "Mary White"}, { "age": 26, "name": "Alex Jones"}, { "age": 19, "name": "Lisa Smith"}]

}

This document will be flattened as we described above, but the result will look like this:

[source,js]

{ "followers.age": [19, 26, 35], "followers.name": [alex, jones, lisa, smith, mary, white]

}

The correlation between `{age: 35}` and `{name: Mary White}` has been lost as each multi-value field is just a bag of values, not an ordered array. This is sufficient for us to ask:

- *Is there a follower who is 26 years old?*

but we can't get an accurate answer to:

- *Is there a follower who is 26 years old* ***and who is called Alex Jones?***

Correlated inner objects, which are able to answer queries like these, are called *nested* objects, and we will discuss them later on in <>.